National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Services des thèses canadiennes

Ottawa, Canada
K1A 0N4

CANADIAN THESES
THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

NL 339 (r. 86/01)

Canada

286

0-315-27017-9

National Library of Canada | Bibliothèque nationale du Canada

Canadian Theses Division | Division des thèses canadiennes

Ottawa, Canada
K1A 0N4

PERMISSION TO MICROFILM — AUTORISATION DE MICROFILMER

• Please print or type — Écrire en lettres moulées ou dactylographier

Full Name of Author — Nom complet de l'auteur

ANTOINE P. VERHEIJEN

Date of Birth — Date de naissance

JULY 26, 1955

Country of Birth — Lieu de naissance

THE NETHERLANDS

Permanent Address — Résidence fixe

3544 12A AVE.
EDMONTON, ALBERTA
T6L 3L5

Title of Thesis — Titre de la thèse

EVALUATION OF DIAGONAL PADE ALGORITHMS

University — Université

UNIVERSITY OF ALBERTA

Degree for which thesis was presented — Grade pour lequel cette thèse fut présentée

MASTER OF SCIENCE

Year this degree conferred — Année d'obtention de ce grade

1983

Name of Supervisor — Nom du directeur de thèse

STANLEY CABAY

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE NATIONALE DU CANADA de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur.

Date

Oct. 17/83

Signature

Antoine Verheijen

NL-91 (4/77)

THE UNIVERSITY OF ALBERTA

Evaluation of Diagonal Pade Algorithms

by

© Antoine Verheijen

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF Master of Science

Department of Computing Science

EDMONTON, ALBERTA

Fall, 1983

MICRO

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR Antoine Verheijen

TITLE OF THESIS Evaluation of Diagonal Pade Algorithms

DEGREE FOR WHICH THESIS WAS PRESENTED Master of Science

YEAR THIS DEGREE GRANTED Fall, 1983

Permission is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(SIGNED) Antoine Verheijen

PERMANENT ADDRESS:

3544 12A Avenue

Edmonton, Alberta

T6L 3L5

DATED October 15 1983

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled Evaluation of Diagonal Pade Algorithms submitted by Antoine Verheijen in partial fulfilment of the requirements for the degree of Master of Science in Computing Science.

.....

Supervisor

.....

.....

.....

Date October 13, 1983

Abstract

Four algorithms are presented for computing symbolic diagonal Pade approximants for a power series whose coefficients lie in a field. Precise bounds are developed for the worst case cost of each of the four algorithms using both classical and fast power series and polynomial arithmetic. Following this, empirical CPU timing results, as obtained for each of the algorithms, are given. Based on both the theoretical and empirical costs produced, it is shown that one method, using classical arithmetic, is superior for obtaining approximants of low degree and that a different method, using fast arithmetic, becomes superior for larger problems. Estimates for the crossover in superiority between the two methods are obtained.

Table of Contents

Chapter	Page
1. Introduction	1
1.1 Pade Approximants	4
2. Polynomial and Power Series Arithmetic	14
2.1 The Classical Algorithms	15
2.2 The Fast Fourier Transform	20
2.3 Fast Multiplication and Division	30
3. Algorithms for Computing Pade Approximants	39
3.1 The AD Algorithm	40
3.2 The MD Algorithm	56
3.3 The DIAG Algorithm	61
3.4 The OFFDIAG Algorithm	69
4. Practical Analysis	77
4.1 Selection of the Coefficient Field	77
4.2 Implementation Details	79
4.3 Empirical Results	82
4.4 Conclusions	87
4.5 Thoughts for Future Research	90
References	96
Appendix A: Determining Theoretical Cost Estimates	100
Appendix B: Empirical CPU Time Values	104

List of Tables

Table	Page
1. CPU Times using Classical Arithmetic	104
2. CPU Times using Fast Arithmetic	105
3. Storage Manipulation Times using Classical Arithmetic	106
4. Storage Manipulation Times using Fast Arithmetic	107

List of Figures

Figure Page

1. Empirical CPU Time Graph 84

2. Theoretical Cost Graph 85

1. Introduction

Over the last few years, a renewed interest has been expressed in the computation of Pade approximants for a power series, which represents one approach to converting a power series to rational form, as outlined shortly. Pade approximants have held a long and established place in mathematical analysis and theoretical physics, as indicated by Gragg [12] and Geddes [10,11]. However, this lastest interest has been generated from within the study of symbolic computer algebra by the desire to be able to efficiently manipulate rational functions.

The most straightforward approach for dealing with rational functions, that of retaining both the numerator and the denominator as separate entities, is susceptible to unpredictable growth in the numerator and the denominator. In particular, performing arithmetic on rational functions has a tendency to dramatically increase the sizes (in terms of their degrees) of the numerator and denominator of the result. This makes it necessary either to retain and work with undesirably large values, or to continually compute polynomial greatest common divisors in order to keep the rational function in reduced form. For some (perhaps all) applications, a possible alternative is available in the form of truncated power series.

The idea behind the use of power series for this purpose is that the rational function is first converted to a truncated power series of a predetermined number of terms.

The desired calculations are then performed on this power series with an appropriate number of terms being retained. Finally, the resulting power series is (are) converted back into rational form.

The transformation from rational form to truncated power series form is relatively easy to perform, for example, by applying power series division (described in Chapter 2) to the numerator and denominator polynomials. However, the reverse transformation is not as easy to accomplish. It is in this area that the usefulness of Pade approximants becomes apparent. To date, Pade approximation techniques represent the most promising method for converting a rational function from truncated power series form back to rational form.

The recent work on Pade approximants has resulted in the development of three fast algorithms that reduce the complexity of computing diagonal Pade approximants (defined in the next section) to $O(n\log^2 n)$ from the previous best of $O(n^2)$ as achieved by the classical algorithms. It is the purpose of the present research to perform a comparative evaluation of these three algorithms along with the best of the classical methods. The analysis is carried out at both the abstract and the practical level, but it is the practical results which form the major contribution of this study. Although two of the algorithms are designed to handle more general cases, and are presented in their more general form, only the computation of Pade approximants which lie

along the main diagonal of the Pade table is considered. This is not limiting in any practical sense since it is shown that algorithms for computing diagonal approximants can easily be used to obtain any arbitrary Pade approximant for a power series. A further limitation imposed in the present study is that all polynomials and power series are assumed to be univariate with coefficients lying in a field.

No mention is made throughout this presentation of any classical method for computing Pade approximants other than the one mentioned above. The reason for this is twofold. To begin with, the classical method that is included herein is 2/3 the cost of the next best classical method. Furthermore, almost all of the other classical algorithms fail for power series that are abnormal (as defined in the next section). Details of these other approaches can be found in Gragg [12] and Geddes [10,11].

Throughout this presentation, some familiarity with the basic concepts of abstract algebra is assumed. Should such knowledge be lacking, it is advised that a text on abstract algebra, such as Herstein [13] or Lipson [19], be consulted to obtain a rudimentary background in the preliminary aspects of ring and field theory.

An effort has been made to keep this discussion as complete as possible in an attempt to reduce the need to consult the various references. To meet this objective, a substantial amount of existing theory is included in this presentation. In particular, although sometimes stated in a

different form in order to produce a unified approach, the algorithms, theorems, and proofs presented herein are not unique to the present discussion and can all be found in the references listed at the end of this presentation. However, because simply knowing the order of complexity of the algorithms is not completely sufficient for performing a practical comparison, more precise worst case cost estimates are developed in this presentation than are available elsewhere in the literature.

In the abstract presentation of any polynomial or power series algorithm in this discussion, all theoretical cost estimates are derived in terms of unit operations, or coefficient operations. Such operations include the addition, subtraction, and multiplication of two elements from the coefficient field. It is assumed that the cost of each of these three types of operations is the same.

1.1 Pade Approximants

A univariate polynomial or power series is defined to be an expression of the form

$$A(x) = \sum_{k=0}^{n} a_k x^k$$

where n is a specific integer value for polynomials and n is ∞ for power series. The a_k values, referred to as the coefficients of the polynomial or power series, are selected from a specific algebraic system. For present purposes, this system is assumed to be a field, known as the coefficient

field. By convention, it is assumed that $a_k = 0$ for all $k > n$. The value x is the indeterminate of the polynomial or power series and is a variable whose domain is the same as the coefficient field.

Because it is not possible, in practice, to retain an infinite number of terms, power series are often truncated to a specified number of terms. This makes truncated power series look very much like polynomials. A difference does still exist, however, in the way that the division operation is defined for these two sets. The details of this difference is deferred until Chapter 2, where the general concepts of polynomial and power series arithmetic are fully described.

A somewhat non-standard convention is adopted for denoting polynomials and power series throughout this presentation. If A(x) is a polynomial or truncated power series, then A denotes the vector of coefficients of A(x). Since all polynomials and power series considered in this discussion are assumed to be univariate, the vector of coefficients completely describes the polynomial or power series. Therefore, because it results in less cluttered notation, the vector form of denoting polynomials and truncated power series is used. Unless otherwise indicated, the kth coefficient of polynomial or power series A is denoted by a_k while the indeterminate of all polynomials and power series is assumed to be x.

Two functions are used to indicate the magnitude of a polynomial or truncated power series.

Definition 1.1.1

Let A be a non-zero polynomial or truncated power series. Then

1) $\deg(A) = \max\{k \text{ such that } a_k \neq 0\}$ = degree of A,

2) $\operatorname{ord}(A) = \min\{k \text{ such that } a_k \neq 0\}$ = ordinate of A.

By convention, deg(0) is $-\infty$ while ord(0) is ∞. Given this definition, a zero-ordinate polynomial or power series is one whose ordinate is zero. Some properties of the degree and ordinate functions, which are required later, now follow:

Theorem 1.1.2

Let A,B be two non-zero polynomials or truncated power series. Then

1) $\deg(A+B) \leq \max\{\deg(A),\deg(B)\}$,

2) $\deg(AB) \leq \deg(A)+\deg(B)$,

3) $\operatorname{ord}(A+B) \geq \min\{\operatorname{ord}(A),\operatorname{ord}(B)\}$,

4) $\operatorname{ord}(AB) = \operatorname{ord}(A)+\operatorname{ord}(B)$.

These properties are a direct consequence of the definitions of the two functions and the definitions of the addition and multiplication operations for polynomials and power series. When dealing only with polynomials, the second inequality in this theorem becomes an equality. Note also that fourth

property as applied to truncated power series assumes that the product of A and B is not zero within the degree of truncation.

When considering the division operation for polynomials, an unfamiliar notational convention is adopted in this presentation. As with the integers, the division of two polynomials produces both a quotient and a remainder.

Definition 1.1.3

Let A,B,Q,R be polynomials such that $B \neq 0$, $A = QB+R$, and $\deg(R) < \deg(B)$. Then

1) $\lfloor A/B \rfloor = Q$ = quotient of A/B,
2) $A \bmod B = R$ = remainder of A/B.

Although non-standard, this method of denoting the quotient of the polynomial division operation is not unique to the present discussion as it has been used in such works as Aho, Hopcroft, and Ullman [1]. The mod operation should be a familiar concept from abstract algebra.

Since the division of two power series only produces a quotient, a similar method of distinguishing the quotient of a power series division is not required. Furthermore, a mod operator of the type defined for polynomials does not exist for power series. However, a similar mod operation is defined for power series:

Definition 1.1.4

Let A be an arbitrary power series. Then

$$A \bmod x^n = \sum_{k=0}^{n-1} a_k x^k.$$

Note that the second operand for the mod operator as applied to power series is always a power of the indeterminate. Thus, the mod operation for power series provides a convenient mechanism for indicating the degree of truncation for a truncated power series.

One final algebraic system needs to be defined. A rational function is an expression of the form A/B where A, the numerator, and B, the denominator, are polynomials. Equality for rational functions is defined as follows:

Definition 1.1.5

Let A/B, C/D be two rational functions. Then A/B = C/D iff AD = BC.

Note that this definition makes no assertions about the equality of the numerators or the denominators. The two basic arithmetic operations performed for rational functions are given below:

Definition 1.1.6

Let A/B, C/D be two rational functions. Then

1) (A/B)+(C/D) = (AD+BC)/(BD),

2) (A/B)(C/D) = (AC)/(BD).

Finally, a rational function, A/B, is said to be in reduced form if gcd(A,B) = 1.

With this background information given, it is now possible to introduce the concept of a Pade approximant.

Definition 1.1.7

Let A be a power series. Then the rational function P/Q is an (m,n) Pade approximant for A if

1) $AQ-P = 0 \mod x^{m+n+1}$,

2) $\deg(P) \leq m$,

3) $\deg(Q) \leq n$.

One way of interpreting this definition is that, if a rational function, P/Q, with ord(Q) = 0, is an (m,n) Pade approximant for a power series, A, then the application of power series division to P and Q produces a power series whose first m+n+1 coefficients are the same as the corresponding coefficients of A. A rational function is referred to as a reduced (m,n) Pade approximant if it is equivalent to the reduced form of the rational function represented by the (m,n) Pade approximant. Note that by this definition, a reduced Pade approximant is not a Pade approximant according to the definition given above; it is simply the reduced rational form of the given Pade approximant.

The complete set of Pade approximants for a given power series are normally organized into a table, called the Pade table for that power series, where the (m,n)th entry of the table corresponds to the (m,n) Pade approximant for the power series. A power series is said to be normal, with

respect to its Pade table, if every approximant in the Pade table is not equal to any other approximant in the table. A power series is (m,n)-normal if its (i,j) Pade approximants are distinct from each other for all $0 \leq i \leq m$ and $0 \leq j \leq n$. A power series is normal along a diagonal or anti-diagonal of its Pade table if all approximants along that diagonal or anti-diagonal are distinct.

If a Pade approximant for a power series lies on the main diagonal of the Pade table for that power series, then that approximant is referred to as a diagonal (n,n) Pade approximant for the power series. A handy definition associated with diagonal Pade approximants is given as follows:

Theorem 1.1.8

> Given a diagonal (n,n) Pade approximant for a power series, the value of n is defined to be the degree of conversion for that Pade approximant.

This definition provides a useful means of referring to the position of a diagonal Pade approximant in the Pade table.

An interesting result concerning the Pade table for an abnormal power series (one which is not normal) is contained in the following theorem.

Theorem 1.1.9

> Let A be a power series. Let P/Q be a reduced rational function with deg(P) = m and deg(Q) = n such that

$AQ-P = 0 \bmod x^{m+n+1}$. Let $k \geq 0$ be the largest integer such that $AQ-P = 0 \bmod x^{m+n+k+1}$. Finally, let R_{ij} be the (i,j) Pade approximant of A for $m \leq i \leq m+k$ and $n \leq j \leq n+k$. Then $R_{ij} = P/Q$.

The proof of this result is given by Gragg [12], whose article constitutes an excellent discussion of the concept of Pade approximants in general. The implication of Theorem 1.1.9 is that equal Pade approximants for an abnormal power series occur in rectangular blocks in the Pade table for that power series.

Another useful feature of Pade approximants is presented as the following result.

Theorem 1.1.10

Let A,B,C be power series such that $B = A \bmod x^{m-n}$ and $C = (A-B)/x^{m-n}$ for two integers $m \geq n$. Now let $\bar{P}/\bar{Q}$ be the (n,n) Pade approximant for C. Then the (m,n) Pade approximant for A is the rational function P/Q where $P = B\bar{Q}+\bar{P}x^{m-n}$ and $Q = \bar{Q}$.

Proof:

$$\begin{aligned} AQ-P &= A\bar{Q}-(B\bar{Q}+\bar{P}x^{m-n}) \\ &= (A-B)\bar{Q}-\bar{P}x^{m-n} \\ &= (C\bar{Q}-\bar{P})x^{m-n} \\ &= (0 \bmod x^{2n+1})x^{m-n} \\ &= 0 \bmod x^{m+n+1}. \end{aligned}$$

Also, $\deg(P) = \deg(B\bar{Q}+\bar{P}x^{m-n})$

$$\leq \max\{\deg(B\bar{Q}), \deg(\bar{P})+m-n\}$$

$$\leq \max\{m-1, m\}$$

$$= m$$

and $\deg(Q) = \deg(\bar{Q}) \leq n$.

Thus, P/Q is the (m,n) Pade approximant for A.

Q.E.D.

This result shows that to compute any Pade approximant on or below the main diagonal of the Pade table for a power series, it is only necessary to have available a method that computes approximants along the main diagonal of a Pade table.

To complete this abstract discussion of Pade approximants, one more property of such approximants is presented. In order to present this result, the definition of Pade approximants is first modified slightly.

Definition 1.1.11

Let A,B,C be power series such that A = B/C. Then the rational function P/Q is an (m,n) Pade approximant for A if

1) $BQ-CP = 0 \mod x^{m+n+1}$,

2) $\deg(P) \leq m$,

3) $\deg(Q) \leq n$.

Note that if C = 1, then this new definition is the same as that given earlier. Using this new definition, the following relation between the Pade approximants of two power series which are inverses of each other can be given.

Theorem 1.1.12

Let A,B,C be power series such that A = B/C and ord(A) = 0. Let P/Q be an (m,n) Pade approximant for A. Then Q/P is an (n,m) Pade approximant for power series $A^{-1} = C/B$.

The proof of this theorem is trivial, given the extended definition of Pade approximants. This result is given using the more general definition of Pade approximants because it is required in that form later. A benefit of this result is that, coupled with Theorem 1.1.10, it indicates that the computation of any Pade approximant for a power series requires only the availability of a method for computing main diagonal Pade approximants.

2. Polynomial and Power Series Arithmetic

Of primary concern in the analysis of any algebraic algorithm are the methods used to perform the basic arithmetic operations, since these have such a direct effect on the complexity of the problem. Two techniques are commonly considered when discussing polynomial and power series arithmetic: the classical algorithms and the fast algorithms. Even though the asymptotic complexity of the latter are known to be superior, both approaches are presented in this chapter for reasons which will become apparent later.

Before proceeding, a notational convention adopted in this chapter is restated. All polynomials and power series are assumed to be univariate with coefficients belonging to a field. If A(x) is a polynomial or power series, then A denotes the vector of coefficients of A(x) while a_k denotes the kth coefficient of A(x). On the other hand, if V is a vector, then V(x) is the polynomial or power series whose coefficients are the elements of V. Taking advantage of this duality, only the vector form notation is used for polynomials or power series, since it is less cumbersome in appearance and does not result in any ambiguities, while x is used to denote the indeterminate of all polynomials and power series throughout.

A couple of definitions are also repeated. Firstly, a zero-ordinate power series is a power series whose constant term is non-zero. Secondly, the costs of polynomial or power

series algorithms are determined in terms of unit operations, which are defined to be the operations of addition, subtraction, or multiplication when performed on two elements from the coefficient field.

2.1 The Classical Algorithms

The easiest arithmetic operations to perform on polynomials and power series are those of addition and subtraction. If A and B are two polynomials, then their sum or difference, $C = A \pm B$, is calculated as follows:

Algorithm 2.1.1: Polynomial Addition and Subtraction

Input: A,B are polynomials.
Output: $C = A \pm B$.

1) $n = \max[\deg(A)+1, \deg(B)+1]$.
2) For $i = 0$ until $n-1$ do begin
3) $\quad c_i = a_i \pm b_i$.
end.

It can be observed that both polynomial addition and subtraction each require n operations involving elements from the coefficient field. If A and B are two power series instead, then the value of n becomes the number of terms retained when using truncated power series and the complexity of addition and subtraction is the same.

The product, $C = AB$, of two arbitrary polynomials or power series, A and B, is defined by the Cauchy product as

follows:

Algorithm 2.1.2: Polynomial/Power Series Multiplication

Input: A,B are polynomials or power series.

Output: C = AB.

1) n = deg(C)+1.
2) For i = 0 until n-1 do begin
3) $c_i = 0$.
4) For j = 0 until i do begin
5) $c_i = c_i + a_j b_{i-j}$.
end.
end.

For polynomials, n = deg(A)+deg(B)+1 whereas for power series, n is the number of terms retained when using truncated power series. The complexity of this algorithm, as determined by step 5, is n^2+n unit operations.

For polynomials, the complexity of Algorithm 2.1.2 can be improved by using the fact that

$$a_j b_{i-j} = 0 \quad \text{if } j > \deg(A) \text{ or } i-j > \deg(B).$$

Eliminating these terms from the previous algorithm, and rearranging, produces the following result:

Algorithm 2.1.3: Polynomial Multiplication

Input: A,B are polynomials.

Output: C = AB.

1) For i = 0 until deg(A)+deg(B) do $c_i = 0$.
2) For i = 0 until deg(A) do begin
3) For j = 0 until deg(B) do begin
4) $c_{i+j} = c_{i+j} + a_i b_j$.
end.
end.

The complexity of this algorithm is 2[deg(A)+1][deg(B)+1], which is easily verified as an improvement over the previous result. A similar improvement is not possible, however, for arbitrary power series multiplication because, in general, deg(A) = deg(B) = n-1 for any arbitrary truncated power series, A and B.

Up to this point, the arithmetic operations for polynomials and power series have been very similar. The same is not true for the operation of division. This is due to the fact that the division of two polynomials produces both a quotient and a remainder while the division of two power series produces only a quotient. For two arbitrary polynomials, A and B, the quotient, Q, and the remainder, R, are produced as follows:

Algorithm 2.1.4: Polynomial Division

Input: A,B are polynomials such that deg(B)≤deg(A).
Output: Q,R such that A = QB+R, deg(R)<deg(B).

1) R=A, m=deg(A), n=deg(B).
2) $c = b_n^{-1}$.

3) For i = m-n until 0 do begin

4) $q_i = r_{n+i}c$.

5) $r_{n+i} = 0$.

6) For j = i until n+i-1 do begin

7) $r_j = r_j - q_i b_{j-i}$.

end.

end.

Step 2 requires calculating the inverse of a unit element, the cost of which is dependent on the actual coefficient field used. Step 4 requires 1 operation per execution while steps 6 and 7 combined require 2n operations per execution. Thus, the total cost of polynomial division is $2n(m-n)+m+n+1$ unit operations plus 1 unit inverse computation.

On the other hand, the algorithm for dividing an arbitrary power series, A, by a zero-ordinate power series, B, is a variation of the Cauchy rule.

Algorithm 2.1.5: Power Series Division

Input: A is an arbitrary power series,
B is a zero-ordinate power series.

Output: $Q = AB^{-1} \bmod x^n$.

1) n = deg(Q)+1 = number of terms required.

2) $c = b_0^{-1}$.

3) For i = 0 until n-1 do begin

4) $q_i = a_i$.

5) For j = 0 until i-1 do begin

6) $q_i = q_i - q_j b_{i-j}$.

end.

7) $q_i = q_i c$.

end.

In this algorithm, step 2 requires the calculation of a unit inverse, steps 5 and 6 combined require 2i operations per execution, and step 7 requires 1 unit operation per execution. This brings the total cost of power series division to n^2 unit operations plus 1 unit inverse computation.

A careful examination of the last algorithm shows that not every power series can be a divisor in the division operation. This is because the constant term, b_0, of the divisor need not be non-zero for every power series. If the constant term is zero, then its inverse does not exist so that the division algorithm will not work. This situation can be rectified to some extent so that division will work if the ordinate of the dividend, ord(A), is no less than the ordinate of the divisor, ord(B). Since all power series divisons required by any later algorithms meet the conditions of the above algorithm, this extension is not presented. It is important to remember, however, that this restriction does exist.

Although covered by the previous algorithms, the multiplication and division of an arbitrary polynomial or power series by a power of its indeterminate should be dealt with as separate operations. These operations often do not

have any cost attributed to them, regardless of the size of the values being operated on, because no actual unit arithmetic is required to produce the result. Although the computational model permits this assumption, it is not completely justified in practice since, at best, some manipulation of the coefficients is invariably necessary. Consequently, to retain a more accurate account of the cost of these operations, a different convention is adopted. Given an arbitrary polynomial or power series, A, and a power, k, of the indeterminate, the multiplication and divison of A by the kth power of the indeterminate have the following associated costs:

Operation	Polynomial Cost	Power Series Cost
Multiplication	deg(A)+k+1	n
Division	deg(A)+1	n

where n is the number of terms retained for truncated power series. The polynomial division cost is the cost of producing both the quotient and the remainder. The cost for producing only one of the two results is assumed to be the same as the cost of producing both.

2.2 The Fast Fourier Transform

A polynomial or power series can be represented by either its coefficients, as is the most common practice, or by a vector of values which result when it is evaluated at a specified number of points. The number of points for which

this evaluation is performed is at least one more than the degree of the polynomial or power series. The discrete fast Fourier transform and its inverse provide an efficient method for changing from the former representation to the latter type and back again. This transformation is discussed because it provides the basis for the fast polynomial and power series multiplication and division techniques presented in following section.

Part of the Fourier transform's efficiency derives from the choice of points for which the evaluation is performed. For a polynomial or power series of degree less than n, the points used are the nth roots of unity for the coefficient field.

Definition 2.2.1

An element, w, in an arbitrary field is a primitive (or principal) nth root of unity if

$$w^n = 1 \quad \text{and} \quad w^i \neq 1$$

for all $0 < i < n$. The values w^k for all $k = 0,\ldots,n-1$ are referred to simply as nth roots of unity.

This definition, given by Borodin and Munro [4], differs somewhat from that normally encountered in the literature, as represented, for example, by Aho, Hopcroft, and Ullman [1] or Horowitz and Sahni [14], in that the following property is usually included as part of the definition.

Theorem 2.2.2

Let w be a primitive nth root of unity in an arbitrary field. Then

$$\sum_{i=0}^{n-1} w^{ij} = 0 \qquad \text{for any } j = 1,\ldots,n-1.$$

Proof:

Let $z = w^j$ for any $j = 1,\ldots,n-1$.

$$\begin{aligned}\text{Then } \sum_{i=0}^{n-1} z^i &= (z^n-1)/(z-1)\\ &= [(w^j)^n-1]/(w^j-1)\\ &= [(w^n)^j-1]/(w^j-1)\\ &= 0.\end{aligned}$$

Q.E.D.

The advantage of removing this property from the definition is that without it, it is easier to verify whether or not any given element of a field might be a root of unity.

Now, let A be a polynomial or power series such that deg(A) < n, and let w be a primitive nth root of unity for the coefficient field. Then the forward discrete Fourier transform is defined by the equation

$$v_i = \sum_{k=0}^{n-1} a_k w^{ik} \qquad 0 \le i \le n-1$$

where v_i is the ith element of the vector, V, forming an alternate representation of A. The fast Fourier transform algorithm uses a divide and conquer approach to reduce the problem of computing these elements to two smaller problems of half the complexity. The value used for n by this

algorithm is $n = 2^r$ where $r = \lceil \log[\deg(A)+1] \rceil$. The fast Fourier transform algorithm can then be presented as follows:

Algorithm 2.2.3: FFT(A,V,w,n)

Input: A is a polynomial or power series,
w is an nth primitive root of unity,
$n = 2^{\lceil \log[\deg(A)+1] \rceil}$.

Output: V = the Fourier transform vector for A.

Comment B,C are polynomials or power series,
S,U are vectors.

1) If n = 1 then $v_0 = a_0$ else begin
2) m = n/2.
3) For i = 0 until m-1 do begin
4) $b_i = a_{2i}$, $c_i = a_{2i+1}$.
end.
5) Call FFT(B,S,w^2,m).
6) Call FFT(C,U,w^2,m).
7) d = 1.
8) For i = 0 until m-1 do begin
9) $d' = du_i$.
10) $v_i = s_i + d'$, $v_{i+m} = s_i - d'$.
11) d = dw.
end.
end.

Let T(n) denote the total cost of this algorithm. Steps 1 to

4 are assumed to be cost-free. Steps 5 and 6 each require T(n/2) unit operations. Steps 9, 10, and 11 combined require 4 operations, bringing the cost of the loop starting at step 8 to 2n unit operations. Therefore,

$$T(n) = 2T(n/2)+2n = 2n\log n+n.$$

Verification of the fact that Algorithm 2.2.3 does indeed compute the forward Fourier transform requires the use of a couple of properties related to roots of unity.

Theorem 2.2.4

Let w be a primitive nth root of unity in an arbitrary field and let n = 2m. Then

$$w^{i+m} = -w^i.$$

Proof:

First note that $w^n = w^{2m} = 1$.

Since $w^m \neq 1$ by definition, this means that $w^m = -1$.

Therefore, $w^{i+m} = w^i w^m = w^i(-1) = -w^i$.

Q.E.D.

Theorem 2.2.5

Let w be a primitive nth root of unity in an arbitrary field and let n = km. Then w^k is a primitive mth root of unity for the given field.

Proof:

Note that $(w^k)^m = w^{km} = w^n = 1$.

Let $1 \leq i \leq m-1$.

Then $ki < km = n$.

Therefore, $(w^k)^i = w^{ki} \neq 1$ since w is a primitive nth root of unity.

Consequently, w^k fits the definition of a primitive mth root of unity.

Q.E.D.

The validity of the fast Fourier transform algorithm can now be verified using mathematical induction. First note that the recursive calls to produce vectors S and U rely on w^2 being a primitive (n/2)th root of unity. This is, in fact, guaranteed by the last theorem. Now assume that the algorithm works for n/2. In particular, this assumption is equivalent to accepting S and U as the true forward transforms for B and C, respectively. The inductive step of the verification then goes as follows:

$$v_i = \sum_{k=0}^{n-1} a_k w^{ik} \qquad 0 \leq i \leq m-1$$

$$= \left(\sum_{k=0}^{m-1} a_{2k} w^{2ik}\right) + \left(\sum_{k=0}^{m-1} a_{2k+1} w^{i(2k+1)}\right)$$

$$= \left(\sum_{k=0}^{m-1} b_k w^{2ik}\right) + \left(\sum_{k=0}^{m-1} c_k w^{2ik} w^i\right)$$

$$= s_i + u_i w^i.$$

$$v_{i+m} = \sum_{k=0}^{n-1} a_k w^{(i+m)k} \qquad 0 \leq i \leq m-1$$

$$= \left(\sum_{k=0}^{m-1} a_{2k} w^{(i+m)2k}\right) + \left(\sum_{k=0}^{m-1} a_{2k+1} w^{(i+m)(2k+1)}\right)$$

$$= \left(\sum_{k=0}^{m-1} b_k[-w^i]^{2k}\right) + \left(\sum_{k=0}^{m-1} c_k[-w^i]^{2k+1}\right)$$

$$= \left(\sum_{k=0}^{m-1} b_k w^{2ik}\right) - \left(\sum_{k=0}^{m-1} c_k w^{2ik} w^i\right)$$

$$= s_i - u_i w^i.$$

Therefore, since all elements of V are covered by these two cases, and since they show that, given the initial assumption, the equations used by the algorithm are the same as computing the defining equations, the fast Fourier transform algorithm can be considered valid by the principle of mathematical induction.

As it turns out, the reverse (or inverse) Fourier transform, which computes the coefficients of a polynomial or power series given its values at the nth roots of unity, is just a slight variation of the forward transform. Let A, w, and V be the same as defined for the forward transform. Then the coefficients of A are computed from V using the equation

$$a_i = (1/n) \sum_{k=0}^{n-1} v_k w^{-ik} = n^{-1} \sum_{k=0}^{n-1} v_k (w^{-1})^{ik} \qquad 0 \le i \le n-1.$$

The rightmost side of this equality suggests that the reverse transform can be computed by making direct use of the fast forward transform algorithm with w^{-1} replacing w as the primitive nth root of unity. This, of course, requires that w^{-1} is also a primitive nth root of unity.

Theorem 2.2.6

Let w be a primitive nth root of unity in an arbitrary field. Then w^{-1} is also a primitive nth root of unity for the same field.

Proof:

First observe that $(w^{-1})^n = (w^n)^{-1} = (1)^{-1} = 1$.

Now let $1 \leq i \leq n-1$.

Then, since $w^i \neq 1$, $(w^{-1})^i = (w^i)^{-1} \neq 1$.

Consequently, w^{-1} is a primitive nth root of unity.

Q.E.D.

Therefore, let the values of w and n be the same as for the fast Fourier transform algorithm. Then the reverse transform can be calculated using the algorithm given below.

Algorithm 2.2.7: $FFT^{-1}(A,V,w,n)$

Input: V is a vector of values,
w is an nth primitive root of unity,
$n = 2^r$ for some r.

Output: A = the polynomial or power series, deg(A)<n, whose Fourier transform is V.

1) For i = 0 until n-1 do begin
2) $a_i = v_i$.
end.
3) Call $FFT(A,V,w^{-1},n)$.
4) $d = n^{-1}$.
5) For i = 0 until n-1 do begin

6) $a_i = v_i d$.

end.

The verification of this algorithm is trivial, so the details are omitted. As to the algorithm's complexity, the loop starting as step 1 is cost-free, step 3 has a cost of 2nlogn+n operations, and the loop starting at step 5 has a cost of n operations. The single operation at step 4 is ignored. This gives the reverse transform algorithm a total cost of 2nlogn+2n unit operations.

The algorithms just presented for computing both the forward and the reverse Fourier transforms rely on the use of recursive procedure calls to produce some of their efficiency. While such calls do not affect the theoretical complexity of an algorithm, they do have an associated practical overhead which makes recursive algorithms less desirable than iterative algorithms of the same cost. (In fact, this overhead is charged on all procedure calls since most programming language implementations do not distinguish between recursive and non-recursive procedures.) For this reason, an iterative version of the fast Fourier transform algorithm, based on an evaluation technique developed by Fiduccia [9], is given below.

Algorithm 2.2.8: Iterative FFT(A,V,w,n)

Input: A is a polynomial or power series,
w is an nth primitive root of unity,
$n = 2^{\lceil \log[\deg(A)+1] \rceil}$.

Output: V = the Fourier transform vector for A.

```
1)  For i = 0 until n-1 do begin
2)      v_i = a_i.
    end.
3)  m=n/2, j=0.
4)  For i = 0 until n-2 do begin
5)      If i < j then begin
6)          d = v_i.
7)          v_i = v_j.
8)          v_j = d.
        end.
9)      k = m.
10)     while k < j+1 do begin
11)         j = j-k.
12)         k = k/2.
        end.
13)     j = j+k.
    end.
14) For k = 2 step k until n do begin
15)     d=1, m=k/2.
16)     For j = 1 until m do begin
17)         For i = j-1 step k until n-1 do begin
18)             d' = v_{i+m}d.
19)             v_{i+m} = v_i-d'.
20)             v_i = v_i+d'.
            end.
21)         d = dw^{n/k}.
```

```
        end.
    end.
```

Verification of the validity of this form of the algorithm, and of the fact that it has the same complexity as the recursive form, can be found in Horowitz and Sahni [14], from whom this particular version was obtained, or in Aho, Hopcroft, and Ullman [1], who present a similar version of the iterative approach.

2.3 Fast Multiplication and Division

Fast polynomial and power series multiplication use the fast Fourier transform to convert the problem to another less costly one. In computing the product, C = AB, for two arbitrary polynomials or power series, A and B, this is accomplished as follows:

Algorithm 2.3.1: Fast Polynomial/Power Series Multiplication

Input: A,B are polynomials or power series.
Output: C = AB.

Comment U,V are vectors.

1) $r = \lceil \log[\deg(A)+\deg(B)+1] \rceil$.
2) $n=2^r$, $m=\deg(C)+1$.
3) w = primitive nth root of unity.
4) Call FFT(A,U,w,n).
5) Call FFT(B,V,w,n).

6) For i = 0 until n-1 do begin

7) $u_i = u_i v_i$.

end.

8) Call $FFT^{-1}(C,U,w,n)$.

9) $C = C \bmod x^m$.

For polynomials, deg(C) = deg(A)+deg(B) while for power series, deg(C) is one less than the number of terms retained for truncated power series. It should be noted, therefore, that the last step of the algorithm is only required for power series multiplication. This algorithm is directly based on the following property of the Fourier transform when applied to polynomials.

Theorem 2.3.2

Let A,B be arbitrary polynomials and let F(·) denote the Fourier transform operation. Then

$$F(AB) = F(A)F(B).$$

Proof:

Let C = AB, m = deg(A), and n ≥ deg(A)+deg(B).

$$\sum_{j=i+1}^{n-1} a_j b_{n+i-j} = \left(\sum_{j=i+1}^{m} a_j b_{n+i-j}\right) + \left(\sum_{j=m+1}^{n-1} a_j b_{n+i-j}\right)$$

$$= \left(\sum_{j=i+1}^{m} a_j 0\right) + \left(\sum_{j=m+1}^{n-1} 0 b_{n+i-j}\right)$$

$$= 0.$$

Thus $c_i = \sum_{j=0}^{i} a_i b_{i-j}$

$$= \left(\sum_{j=0}^{i} a_j b_{(n+i-j) \bmod n}\right) + \left(\sum_{j=i+1}^{n-1} a_j b_{n+i-j}\right)$$

$$= \sum_{j=0}^{n-1} a_j b_{(n+i-j) \bmod n}.$$

Now let $F(\cdot)_k$ denote the kth element of the Fourier transform vector.

Then $$F(AB)_k = \sum_{i=0}^{n-1} \left(\sum_{j=0}^{n-1} a_j b_{(n+i-j) \bmod n}\right) w^{ik}$$

$$= \sum_{j=0}^{n-1} a_j \left(\sum_{i=0}^{n-1} b_{(n+i-j) \bmod n} w^{ik}\right)$$

$$= \sum_{j=0}^{n-1} a_j \left(\sum_{r=-j}^{n-1-j} b_{(n+r) \bmod n} w^{(j+r)k}\right)$$

(where r = i-j)

$$= \sum_{j=0}^{n-1} a_j \left[\left(\sum_{r=0}^{n-1-j} b_{(n+r) \bmod n} w^{(j+r)k}\right) + \left(\sum_{r=n-j}^{n-1} b_{r \bmod n} w^{(j+r-n)k}\right)\right]$$

$$= \sum_{j=0}^{n-1} a_j \left(\sum_{r=0}^{n-1} b_r w^{(j+r)k}\right)$$

$$= \left(\sum_{j=0}^{n-1} a_j w^{jk}\right)\left(\sum_{r=0}^{n-1} b_r w^{rk}\right)$$

$$= F(A)_k F(B)_k.$$

Q.E.D.

Turning to the cost of this algorithm, steps 1 to 3 are ignored, steps 4 and 5 each require 2nlogn+n operations (for a total of 4nlogn+2n operations), the loop starting at step 6 takes n operations, and step 8 has a cost of 2nlogn+2n

operations. This brings the complete cost of fast polynomial multiplication to 6nlogn+5n unit operations. The cost for fast power series multiplication has an additional n operations associated with it as a result of step 9, bringing its total cost to 6nlogn+6n unit operations.

A comparison of the fast multiplication costs with those for classical multiplication indicates that the fast multiplication method is not immediately superior. Assuming the worst case situation for classical multiplication, which occurs when deg(A) = deg(B), it can be observed that the crossover for polynomial multiplication occurs when

$$2[\deg(A)+1][\deg(B)+1]-6n\log n-5n \geq 0.$$

This happens somewhere around deg(A) = deg(B) = 55. The crossover for power series is a little higher. If m is the number of terms retained for power series, then the two techniques crossover when

$$m^2+m-6n\log n-6n \geq 0$$

which happens when m = 78. The crossover point presented here for polynomials is a little higher than that produced by Moenck [23]. This discrepency is due to a slightly different approach to accumulating the number of unit operations. In particular, Moenck does not include the operation in step 11 of Algorithm 2.2.3 (the fast Fourier transform) in his counts.

Fast division of both polynomials and power series rely heavily on the ability to perform power series inversion efficiently. A fast method for determining the inverse of a

power series can be obtained through the use of Newton's approximation formula. Let A be a zero-ordinate power series and let n be the number of terms desired in the result. Then A^{-1} can be computed by using:

Algorithm 2.3.3: Fast Power Series Inversion

PSINV(Q,A,n)

Input: A is a zero-ordinate power series,
n = number of terms desired in result.

Output: $Q = A^{-1} \bmod x^n$.

1) $Q_0 = a_0^{-1}$, $m = \lceil \log n \rceil$.
2) For i = 1 until m do begin
3) $Q_i = (2Q_{i-1} - AQ_{i-1}^2) \bmod x^{2^i}$.
end.
4) $Q = Q_m \bmod x^n$.

This algorithm is based on the idea that each successive value for Q_i is a closer approximation to A^{-1} than the previous one. That this is indeed the case is verified as follows:

Let $e_i = A^{-1} - Q_i$ for all $0 \le i \le m$.

Then $e_{k+1} = A^{-1} - Q_{k+1}$

$$= A^{-1} - (2Q_k - AQ_k^2) \bmod x^{2^{k+1}}$$

$$= A^{-1} - [2(A^{-1} - e_k) - A(A^{-1} - e_k)^2] \bmod x^{2^{k+1}}$$

$$= A^{-1} - [2 - A(A^{-1} - e_k)](A^{-1} - e_k) \bmod x^{2^{k+1}}$$

$$= A^{-1}-(1+Ae_k)(A^{-1}-e_k) \mod x^{2^{k+1}}$$

$$= A^{-1}-(A^{-1}-Ae_k^2) \mod x^{2^{k+1}}$$

$$= Ae_k^2 \mod x^{2^{k+1}}.$$

Thus, it can be shown by induction that

$$\text{ord}(e_{k+1}) \geq 2^{k+1}.$$

Therefore, $\text{ord}(e_m) \geq 2^m \geq n$.

Consequently, $AQ = A(A^{-1}-e_m) = 1-Ae_m = 1 \mod x^n$.

The last equation shows that the value of Q is accurate up to the degree of truncation for truncated power series.

In order to determine the cost of the inversion algorithm, let $K = 2^i$ and let $N = 2^m$. Furthermore, assume that fast power series multiplication is being used. Then, using the observation that $\deg(Q_i) = K-1$, step 1 requires a single unit inverse calculation, step 3 requires 18KlogK+31K operations, and step 4 requires N operations. The cost of step 3 is based on the fact that a multiplication of two truncated power series of degree K/2-1, each, and a multiplication of two truncated power series of degree K-1, each, is required. Therefore, the total cost of inversion is bounded by 36NlogN+27N unit operations plus 1 unit inverse computation.

Given Algorithm 2.3.3, it becomes a simple task to develop a fast power series division algorithm. Given an arbitrary power series, A, and a zero-ordinate power series, B, their quotient, Q = A/B, can be obtained, to within n

terms, by:

Algorithm 2.3.4: Fast Power Series Division

Input: A is an arbitrary power series,
B is a zero-ordinate power series.

Output: $Q = AB^{-1} \bmod x^n$.

1) Call PSINV(Q,B,n).

2) $Q = AQ \bmod x^n$.

The proof of this algorithm is trivial. As to its cost, let $N = 2^{\lceil \log n \rceil}$. Then step 1 requires 36NlogN+27N operations plus 1 unit inverse calculation and step 2 requires 6NlogN+6N operations, giving fast power series division a total cost of 42NlogN+33N unit operations plus 1 unit inverse calculation. As with multiplication, the fast method is not immediately superior. This time, the crossover occurs when $n^2-42N\log N-33N \geq 0$, which happens at n = 682.

Fast division of two polynomials is a little more complex. If A and B are two arbitrary polynomials, then the following algorithm produces the quotient, Q, and remainder, R, which result from the division of A by B.

Algorithm 2.3.5: Fast Polynomial Division

Input: A,B are polynomials such that deg(B)≤deg(A).

Output: Q,R such that A = QB+R, deg(R)<deg(B).

Comment C,D,E are polynomials.

1) $k = \lceil \log[\deg(A)-\deg(B)+1] \rceil$.

2) $n = 2^k$.

3) For i = 0 until deg(B) do begin

4) $\quad c_i = b_{\deg(B)-i}$.

end.

5) Call PSINV(D,C,n).

6) For i = 0 until n-1 do begin

7) $\quad e_i = d_{n-i-1}$.

end.

8) $Q = \lfloor AE/x^{\deg(B)+n-1} \rfloor$.

9) R = A-QB.

The correctness of this algorithm is proven as follows:

$DC = 1 \bmod x^n = Fx^n+1$ where $\deg(F) < \deg(B)$.

Then $EB = x^{\deg(B)+n-1}+G$ where $\deg(G) \leq \deg(F)$.

Thus $R = A - QB$

$= A - \lfloor (AEB)/x^{\deg(B)+n-1} \rfloor$

$= A - \lfloor [A(x^{\deg(B)+n-1}+G)]/x^{\deg(B)+n-1} \rfloor$

$= \lfloor AG/x^{\deg(B)+n-1} \rfloor$.

Now note that $\deg(A) \leq \deg(B)+n-1$ and

$\deg(AG) < \deg(A)+\deg(B)$.

Then $\deg(R) < \deg(A)-n+1 \leq \deg(B)+n-1-n+1 = \deg(B)$.

Consequently, A = QB+R where deg(R) < deg(B).

This shows that the values computed for Q and R are the appropriate quotient and remainder for polynomial division. As an aside, it should be pointed out that the operations performed in steps 3 to 7 are, in fact, just a different

form of the polynomial reciprocal procedure given by Aho, Hopcroft, and Ullman [1].

Now let $N_1 = 2^{\lceil \log[\deg(A)+n] \rceil}$ and $N_2 = 2^{\lceil \log[\deg(A)] \rceil}$. Assume that fast polynomial multiplication is being used. Then step 5 of the fast polynomial division algorithm has a cost of $42n\log n+33n$ operations plus 1 unit inverse computation, step 8 requires $6N_1\log N_1+5N_1+\deg(A)+n$ operations, and step 9 requires $6N_2\log N_2+5N_2+\deg(A)$ operations. The remaining steps are assumed to be cost-free. Thus the total cost of fast polynomial division becomes $42n\log n+34n+6N_1\log N_1+5N_1+6N_2\log N_2+5N_2+2\deg(A)$ unit operations plus 1 unit inverse calculation. For determining the crossover point for classical and fast polynomial division, assume that $\deg(A) = 2\deg(B)$. This particular choice is made because it represents the worst case for the polynomial divisions encountered in the Pade algorithms of the next chapter. The cost of fast division under these conditions collapses to $72n\log n+89n+4\deg(B)$ unit operations plus a single unit inverse computation, which becomes cheaper than classical division at the point where

$$2[\deg(B)]^2-\deg(B)-72n\log n-89n \geq 0.$$

This happens when $\deg(B) = 951$.

3. Algorithms for Computing Pade Approximants

Having provided some background information for handling polynomials and power series via computer, it is now possible to turn to the discussion of algorithms for computing diagonal Pade approximants for a power series. The four algorithms considered in this chapter fall into two categories: one which computes approximants along the anti-diagonals of the Pade table, and three which compute them along the forward diagonals.

As with the previous chapter, the coefficient vector form of denoting univariate polynomials and power series is used, with x being used as the indeterminate. It should also be remembered that a unit operation in a polynomial or power series algorithm refers to the operations of addition, subtraction, and multiplication when performed on two elements from the coefficient field.

A few further definitions are also required by the discussion in this chapter. A power series is said to be normal along a diagonal or anti-diagonal of its Pade table if all approximants along that diagonal or anti-diagonal are distinct. A power series is (m,n)-normal if all (i,j) Pade approximants are distinct for $0 \leq i \leq m$ and $0 \leq j \leq n$. A polynomial sequence is normal if the degree of each successive polynomial in the sequence is exactly one less than the degree of the previous one.

Because two forms of polynomial or power series arithmetic can be used for the various algorithms in this

chapter, one final convention is adopted when determining the initial cost estimates of each algorithm. The function $C_M(m,n)$ is used to denote the cost of multiplying an m-degree polynomial or power series by an n-degree polynomial or power series while the function $C_D(m,n)$ denotes the cost of dividing an m-degree polynomial or power series by a polynomial or power series of degree n. A similar function is not required for addition and subtraction since only one form of these operations exist. The distinction between polynomials and power series need not be made in the notation since the context of each algorithm indicates which entities are being operated on.

3.1 The AD Algorithm

The anti-diagonal approach to computing Pade approximants is based on some observations made by McEliece and Shearer [20] concerning the extended Euclidean algorithm for computing polynomial GCDs. Let A_0 and A_1 be two polynomials such that $\deg(A_1) < \deg(A_0)$. Then the extended Euclidean algorithm is defined as follows:

Algorithm 3.1.1: Extended Euclidean Algorithm

Input: A_0, A_1 are polynomials such that $\deg(A_1)<\deg(A_0)$.
Output: A_k, S_k, T_k such that $S_kA_0+T_kA_1 = A_k = \gcd(A_0,A_1)$.

Comment A_i, S_i, T_i, Q_i are polynomials for all i.

1) $S_0=1$, $S_1=0$, $T_0=0$, $T_1=1$, $i=1$.

2) While $A_{i-1} \bmod A_i \neq 0$ do begin

3) $Q_i = \lfloor A_{i-1}/A_i \rfloor$, $A_{i+1} = A_{i-1} \bmod A_i$.

4) $S_{i+1} = S_{i-1} - S_i Q_i$, $T_{i+1} = T_{i-1} - T_i Q_i$.

5) $i = i+1$.

end.

The sequence $\{A_i\}$ for $0 \leq i \leq k$ is referred to as the polynomial remainder sequence for A_0 and A_1. This algorithm exhibits three very valuable properties:

Theorem 3.1.2

Let A_0, A_1 be two polynomials with $\deg(A_1) < \deg(A_0)$. Then

1) $S_i A_0 + T_i A_1 = A_i$ $\quad 0 \leq i \leq k$

2) $\deg(T_i) + \deg(A_{i-1}) = \deg(A_0)$ $\quad 1 \leq i \leq k$

3) $T_i A_{i-1} - T_{i-1} A_i = (-1)^i A_0$ $\quad 1 \leq i \leq k$

for all A_i, S_i, T_i computed by the extended Euclidean algorithm.

Each of these three properties can be proven using mathematical induction, the details of which are omitted.

At this point, an interesting result, as presented by McEliece and Shearer [20], can be given.

Theorem 3.1.3

Let A_0, A_1 be two polynomials with $\deg(A_1) < \deg(A_0)$. Let m,n be two integers such that $m \geq \deg[\gcd(A_0, A_1)]$ and $m+n = \deg(A_0)-1$. Then there exist unique A_i, T_i

computed by the extended Euclidean algorithm such that

$$\deg(A_i) \leq m, \qquad \deg(T_i) \leq n$$

for some $0 \leq i \leq k$.

Proof:

Let i be the index such that $\deg(A_i) \leq m$ and $\deg(A_{i-1}) \geq m+1$.

Then $\deg(T_i) = \deg(A_0)-\deg(A_{i-1}) \leq m+n+1-m-1 = n$,

and $\deg(T_{i+1}) = \deg(A_0)-\deg(A_i) \geq m+n+1-m = n+1$.

Thus, A_i and T_i exist and are unique.

Q.E.D.

This theorem shows that the extended Euclidean algorithm provides a mechanism for computing Pade approximants for a power series. Let $A_0 = x^{m+n+1}$ and let A_1 be a power series truncated to m+n+1 terms, for two integers m and n. Applying the extended Euclidean algorithm to A_0 and A_1 until $\deg(A_i) \leq m$ produces the polynomials A_i, S_i, and T_i such that

$$S_iA_0 + T_iA_1 = A_i \quad \Rightarrow \quad T_iA_1 = A_i \mod A_0$$
$$\Rightarrow \quad T_iA_1 - A_i = 0 \mod x^{m+n+1}$$

where $\deg(A_i) \leq m$ and $\deg(T_i) \leq n$. This indicates that A_i/T_i is the (m,n) Pade approximant for power series A_1. It is a trivial matter, therefore, to verify that for any arbitrary power series, B, and integer, d, the extended Euclidean algorithm can be used to obtain all the Pade approximants for B along the anti-diagonal in the Pade table whose indices, m and n, satisfy the equation m+n = d-1.

Having developed this technique for computing Pade approximants, the problem now arises as to determining the best method for computing the polynomials A_i, S_i, and T_i, defined by the extended Euclidean algorithm. The most efficient method, in terms of asymptotic cost, for producing these polynomials is a GCD algorithm first presented (in its most general form) by Moenck [21,22]. To facilitate the presentation of this algorithm, the following definition is made.

Definition 3.1.4

Let $\{Q_i\}$ be the sequence of quotients computed by the extended Euclidean algorithm for $0 < i < k$. Then the matrices $M_{i,j}$ are defined as follows:

1) $M_{i,i} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ $\qquad 0 \leq i \leq k,$

2) $M_{i,j} = \begin{bmatrix} 0 & 1 \\ 1 & -Q_i \end{bmatrix} * \ldots * \begin{bmatrix} 0 & 1 \\ 1 & -Q_{j+1} \end{bmatrix}$ $\qquad 0 < j < i < k.$

To make effective use of these matrices, the following properties are also required.

Theorem 3.1.5

Let $M_{i,j}$ be defined as above for all $0 \leq j \leq i \leq k$. Then

1) $M_{i,j} = M_{i,r} M_{r,j}$ $\qquad j \leq r \leq i$

$$2)\quad \begin{bmatrix} A_i \\ A_{i+1} \end{bmatrix} = M_{i,j} \begin{bmatrix} A_j \\ A_{j+1} \end{bmatrix} \qquad 0 \le j \le i \le k$$

$$3)\quad M_{i,0} = \begin{bmatrix} S_i & T_i \\ S_{i+1} & T_{i+1} \end{bmatrix} \qquad 0 \le i \le k$$

where A_i, S_i, T_i are as defined by the extended Euclidean algorithm.

The first property is a direct consequence of the definition while the other two properties can be proven by trivial inductive arguments.

The efficient GCD algorithm mentioned above actually consists of two separate algorithms: one which computes the two polynomials which lie at the exact middle of the remainder sequence produced by the extended Euclidean algorithm, and another which uses the first one to compute the rest of the polynomials in the remainder sequence. Only the first one is presented here since it is shown later that the other is not required for computing diagonal Pade approximants for a power series. Let A_0 and A_1 be two polynomials such that $\deg(A_1) < \deg(A_0)$. Then the EMGCD algorithm, given below, computes the matrix

$$M = \begin{bmatrix} A_i & S_i & T_i \\ A_{i+1} & S_{i+1} & T_{i+1} \end{bmatrix}$$

where $\deg(A_{i+1}) < \lceil \deg(A_0)/2 \rceil \le \deg(A_i)$ and A_i, A_{i+1}, S_i, S_{i+1}, T_i, and T_{i+1} are as defined by the extended Euclidean algorithm.

Algorithm 3.1.6: EMGCD(M, A_0, A_1)

Input: A_0, A_1 are polynomials such that $\deg(A_1) < \deg(A_0)$.

Output: $M = \begin{bmatrix} A_i & S_i & T_i \\ A_{i+1} & S_{i+1} & T_{i+1} \end{bmatrix}$

where $S_i A_0 + T_i A_1 = A_i$, $S_{i+1} A_0 + T_{i+1} A_1 = A_{i+1}$,

$\deg(A_{i+1}) < \lceil \deg(A_0)/2 \rceil \leq \deg(A_i)$.

Comment $B_0, B_1, C_0, C_1, D, E, F, Q, G_0, G_1, H_0, H_1$ are polynomials,

$\bar{M}_1, M_1, \bar{M}_2, M_2$ are matrices.

1) If $A_1 = 0$ or $\deg(A_1) < \deg(A_0)/2$ then $M = \begin{bmatrix} A_0 & 1 & 0 \\ A_1 & 0 & 1 \end{bmatrix}$

else begin

2) $m = \lceil \deg(A_0)/2 \rceil$.

3) $B_0 = \lfloor A_0/x^m \rfloor$, $C_0 = A_0 \bmod x^m$.

4) $B_1 = \lfloor A_1/x^m \rfloor$, $C_1 = A_1 \bmod x^m$.

5) Call EMGCD($\bar{M}_1, B_0, B_1$).

6) $M_1 = \bar{M}_1 \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}^T$.

7) $\begin{bmatrix} D \\ E \end{bmatrix} = \bar{M}_1 [x^m \; C_0 \; C_1]^T$.

8) If $\deg(E) < \deg(A_0)/2$ then $M = \left[\begin{bmatrix} D \\ E \end{bmatrix}, M_1 \right]$

else begin

9) $Q = \lfloor D/E \rfloor$, $F = D \bmod E$, $k = 2m - \deg(E)$.

10) $G_0 = \lfloor E/x^k \rfloor$, $H_0 = E \bmod x^k$.

11) $G_1 = \lfloor F/x^k \rfloor$, $H_1 = F \bmod x^k$.

12) Call EMGCD($\bar{M}_2, G_0, G_1$).

13) $$M_2 = \bar{M}_2 \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}^T .$$

14) $$M = \left[\bar{M}_2 [x^k \; H_0 \; H_1]^T \; , \; M_2 \begin{bmatrix} 0 & 1 \\ 1 & -Q \end{bmatrix} M_1 \right].$$

end.

end.

This version of the partial GCD algorithm corresponds to that given by Brent, Gustavson, and Yun [5].

In order to prove the validity of the EMGCD algorithm, a couple of preliminary results must first be given.

Theorem 3.1.7

Let A,B be two polynomials such that deg(B) < deg(A). Let A_1, A_2, B_1, B_2 be polynomials such that $A = A_1 x^k + A_2$, $B = B_1 x^k + B_2$ where $k \leq 2\deg(B) - \deg(A)$, $\deg(A_2) < k$, $\deg(B_2) < k$. Finally, let $Q = \lfloor A/B \rfloor$, $Q_1 = \lfloor A_1/B_1 \rfloor$, $R = A - QB$, $R_1 = A_1 - Q_1 B_1$. Then

1) $Q = Q_1$,

2) $\lfloor R/x^{\deg(A)-\deg(B)+k} \rfloor = \lfloor R_1/x^{\deg(A)-\deg(B)} \rfloor$.

Proof:

Let $R' = A - Q_1 B$

$= (A_1 x^k + A_2) - Q_1 (B_1 x^k + B_2)$

$= (A_1 - Q_1 B_1) x^k + (A_2 - Q_1 B_2)$

$= R_1 x^k + A_2 - Q_1 B_2$.

Now, $\deg(R_1 x^k) = \deg(R_1) + \deg(x^k)$

$$< \deg(B_1)+k$$

$$= \deg(B),$$

and $\deg(Q_1B_2) = \deg(Q_1)+\deg(B_2)$

$$< \deg(A_1)-\deg(B_1)+k$$

$$= \deg(A)-\deg(B)+k$$

$$\leq \deg(A)-\deg(B)+2\deg(B)-\deg(A)$$

$$= \deg(B).$$

Thus, $\deg(R') \leq \max[\deg(R_1x^k),\deg(A_2),\deg(Q_1B_2)]$

$$< \max[\deg(B),k,\deg(B)]$$

$$= \deg(B).$$

Therefore, $A = Q_1B+R'$ where $\deg(R') < \deg(B)$.

Consequently, $Q = Q_1$ by uniqueness of quotients.

Now, for convenience, let $n = \deg(A)-\deg(B)$.

Then $\lfloor R/x^{n+k} \rfloor = \lfloor R'/x^{n+k} \rfloor$

$$= \lfloor (R_1x^k+A_2-Q_1B_2)/x^{n+k} \rfloor$$

$$= \lfloor R_1/x^n \rfloor.$$

Thus, $\lfloor R/x^{\deg(A)-\deg(B)+k} \rfloor = \lfloor R_1/x^{\deg(A)-\deg(B)} \rfloor$.

Q.E.D.

The importance of this theorem, other than displaying an interesting property of the quotients of polynomials, is that it is required for proving the following result.

Theorem 3.1.8

Let A_0,A_1 be two polynomials such that $\deg(A_1) < \deg(A_0)$. Let B_0,B_1,C_0,C_1 be polynomials such that $A_0 = B_0x^k+C_0$, $A_1 = B_1x^k+C_1$ where $\deg(C_0) < k$, $\deg(C_1) < k$. Let $M_{i,0}$ be the matrix of the form

described in Definition 3.1.4 such that

$$M_{i,0}\begin{bmatrix}B_0\\B_1\end{bmatrix} = \begin{bmatrix}B_i\\B_{i+1}\end{bmatrix}$$

where $\deg(B_{i+1}) < \lceil[\deg(A_0)-k]/2\rceil \leq \deg(B_i)$. Then

$$M_{i,0}\begin{bmatrix}A_0\\A_1\end{bmatrix} = \begin{bmatrix}A_i\\A_{i+1}\end{bmatrix}$$

where $\deg(A_{i+1}) < \lceil[\deg(A_0)+k]/2\rceil \leq \deg(A_i)$.

Proof:

The proof of this theorem is by induction.

For convenience, let $n = \deg(A_0)$, $n^* = \deg(A_{j-2})$, $n' = \deg(A_{j-1})$, $n'' = \deg(A_j)$, $r' = \deg(B_{j-1})$, $r'' = \deg(B_j)$.

Assume first that $M_{j-1,0}$ is a matrix of the form described in Definition 3.1.4 such that

$$M_{j-1,0}\begin{bmatrix}B_0\\B_1\end{bmatrix} = \begin{bmatrix}B_{j-1}\\B_j\end{bmatrix}$$

and

$$M_{j-1,0}\begin{bmatrix}A_0\\A_1\end{bmatrix} = \begin{bmatrix}A_{j-1}\\A_j\end{bmatrix}.$$

Assume also that $\lfloor A_{j-1}/x^{n-n^*+k}\rfloor = \lfloor B_{j-1}/x^{n-n^*}\rfloor$ and $\lfloor A_j/x^{n-n'+k}\rfloor = \lfloor B_j/x^{n-n'}\rfloor$.

This implies that $\deg(B_{j-1}) = \deg(A_{j-1})-k$ and $\deg(B_j) = \deg(A_j)-k$.

Assume finally that $\lceil[\deg(A_0)-k]/2\rceil \leq \deg(B_{j-1})$ and $\lceil[\deg(A_0)+k]/2\rceil \leq \deg(A_{j-1})$.

Since $\deg(A_{j-1}) \leq \deg(A_{j-2})$, $n-n^* \leq n-n'$.

Therefore, $\lfloor A_{j-1}/x^{n-n'+k}\rfloor = \lfloor B_{j-1}/x^{n-n'}\rfloor$.

Now suppose that $2\deg(B_j)-\deg(B_{j-1}) < n-n'$.

Then, by substitution,

$$2\deg(B_j)-\deg(A_{j-1})+k < \deg(A_0)-\deg(A_{j-1})$$
$$\Rightarrow \deg(B_j) < \lceil[\deg(A_0)-k]/2\rceil.$$

Also, by adding k to each side,

$$2\deg(B_j)-\deg(B_{j-1})+k < n-n'+k$$
$$\Rightarrow 2\deg(A_j)-\deg(A_{j-1}) < \deg(A_0)-\deg(A_{j-1})+k$$
$$\Rightarrow \deg(A_j) < \lceil[\deg(A_0)+k]/2\rceil.$$

Thus, in this case, the theorem holds with $B_i = B_{j-1}$, $B_{i+1} = B_j$, $A_i = A_{j-1}$, $A_{i+1} = A_j$, and $M_{i,0} = M_{j-1,0}$ for both B_0,B_1 and A_0,A_1 as desired.

Therefore, assume $n-n' \le 2\deg(B_j)-\deg(B_{j-1})$.

Then $\lceil[\deg(A_0)-k]/2\rceil \le \deg(B_j)$.

Also, $n-n'+k \le 2\deg(B_j)-\deg(B_{j-1})+k$
$= 2\deg(A_j)-\deg(A_{j-1})$,
which means that $\lceil[\deg(A_0)+k]/2\rceil \le \deg(A_j)$.

Thus the third assumption holds by induction.

Now let $D_{j-1} = \lfloor A_{j-1}/x^{n-n'+k}\rfloor = \lfloor B_{j-1}/x^{n-n'}\rfloor$ and
$D_j = \lfloor A_j/x^{n-n'+k}\rfloor = \lfloor B_j/x^{n-n'}\rfloor$.

This means that, for some E_{j-1},E_j,F_{j-1},F_j,

$A_{j-1} = D_{j-1}x^{n-n'+k}+E_{j-1}$ where $\deg(E_{j-1}) < n-n'+k$,

$A_j = D_jx^{n-n'+k}+E_j$ where $\deg(E_j) < n-n'+k$,

$B_{j-1} = D_{j-1}x^{n-n'}+F_{j-1}$ where $\deg(F_{j-1}) < n-n'$,

$B_j = D_jx^{n-n'}+F_j$ where $\deg(F_j) < n-n'$.

Let $Q_j = \lfloor D_{j-1}/D_j\rfloor$ and $R_j = D_{j-1}-Q_jD_j$.

Then, by Theorem 3.1.7, $Q_j = \lfloor A_{j-1}/A_j\rfloor = \lfloor B_{j-1}/B_j\rfloor$.

Thus, $A_{j+1} = A_{j-1}-Q_jA_j$ and $B_{j+1} = B_{j-1}-Q_jB_j$.

Also, again by Theorem 3.1.7,

$$\begin{aligned}\lfloor A_{j+1}/x^{n-n''+k} \rfloor &= \lfloor A_{j+1}/x^{n'-n''+n-n'+k} \rfloor \\ &= \lfloor R_j/x^{n'-n''} \rfloor \\ &= \lfloor R_j/x^{r'-r''} \rfloor \\ &= \lfloor B_{j+1}/x^{r'-r''+n-n'} \rfloor \\ &= \lfloor B_{j+1}/x^{n-n''} \rfloor .\end{aligned}$$

Thus, the second assumption holds by induction.

Finally, $\begin{bmatrix} B_j \\ B_{j+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -Q_j \end{bmatrix} \begin{bmatrix} B_{j-1} \\ B_j \end{bmatrix}$

$$= M_{j,j-1} M_{j-1,0} \begin{bmatrix} B_0 \\ B_1 \end{bmatrix}$$

$$= M_{j,0} \begin{bmatrix} B_0 \\ B_1 \end{bmatrix},$$

and $\begin{bmatrix} A_j \\ A_{j+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -Q_j \end{bmatrix} \begin{bmatrix} A_{j-1} \\ A_j \end{bmatrix}$

$$= M_{j,j-1} M_{j-1,0} \begin{bmatrix} A_0 \\ A_1 \end{bmatrix}$$

$$= M_{j,0} \begin{bmatrix} A_0 \\ A_1 \end{bmatrix}.$$

Consequently, the first assumption also holds by induction and the theorem is valid.

Q.E.D.

It is the result in this theorem that forms the basis of the EMGCD algorithm.

The proof of the EMGCD algorithm's correctness can now be given, using induction, as follows:

Let i,j be the indices in the remainder sequence for

A_0 and A_1 such that $\deg(A_{j+1}) < \lceil 3\deg(A_0)/4 \rceil \leq \deg(A_j)$ and $\deg(A_{i+1}) < \lceil \deg(A_0)/2 \rceil \leq \deg(A_i)$.

Assume that the recursive procedure calls produce the proper result.

First, it is shown that $M_1 = M_{j,0}$ for A_0 and A_1.

By the inductive hypothesis,

$$\begin{bmatrix} B_r \\ B_{r+1} \end{bmatrix} = \bar{M}_1 [1\ 0\ 0]^T = M_1 \begin{bmatrix} B_0 \\ B_1 \end{bmatrix}$$

are the polynomials in the remainder sequence for B_0 and B_1 such that $\deg(B_{r+1}) < \lceil \deg(B_0)/2 \rceil \leq \deg(B_r)$.

Then, by Theorem 3.1.8, $M_1 = M_{j,0}$ for A_0 and A_1.

Thus,
$$\begin{bmatrix} A_j \\ A_{j+1} \end{bmatrix} = M_1 \begin{bmatrix} A_0 \\ A_1 \end{bmatrix} = M_1 \begin{bmatrix} B_0 \\ B_1 \end{bmatrix} x^m + M_1 \begin{bmatrix} C_0 \\ C_1 \end{bmatrix}$$
$$= \begin{bmatrix} B_r \\ B_{r+1} \end{bmatrix} x^m + M_1 \begin{bmatrix} C_0 \\ C_1 \end{bmatrix}$$
$$= \begin{bmatrix} D \\ E \end{bmatrix}.$$

where $\deg(A_{j+1}) < \lceil 3\deg(A_0)/4 \rceil \leq \deg(A_j)$.

Next, by definition, $F = A_{j+1}$ and $\begin{bmatrix} 0 & 1 \\ 1 & -Q \end{bmatrix} = M_{j+1,j}$ for A_0 and A_1.

Finally, by an agrument identical to that for M_1, $M_2 = M_{i,0}$ for E and F.

This implies that $M_2 = M_{i,j+1}$ for A_0 and A_1.

Consequently, $M = \left[\begin{bmatrix} A_i \\ A_{i+1} \end{bmatrix}, M_{i,j+1} M_{j+1,j} M_{j,0} \right]$

$$= \left[\begin{bmatrix} A_i \\ A_{i+1} \end{bmatrix} , M_{i,0} \right]$$

$$= \begin{bmatrix} A_i & S_i & T_i \\ A_{i+1} & S_{i+1} & T_{i+1} \end{bmatrix}.$$

Therefore, by induction, the algorithm produces the expected result.

For the purpose of determining the complexity of this algorithm, a number of assumptions are made. First, it is assumed that $\deg(A_1) = \deg(A_0)-1$ and that the remainder sequence for the input polynomials is normal. This represents the worst case (highest cost) situation for this algorithm. Secondly, it is assumed that $n = \deg(A_1)$ is an exact power of 2. This is done for notational convenience mostly, and does not detract from the complexity equation which results, since the same convention is adopted for all Pade algorithms discussed later. The major consequence of this last assumption is that $\lceil n/2^i \rceil = \lfloor n/2^i \rfloor = n/2^i$ for all $2^i \leq n$, which is valuable in reducing the number of terms in the final cost estimate.

Based on these assumptions, the following statements can be made concerning the sizes of various intermediate results:

$\deg(B_0) = n/2$	$\deg(C_0) = n/2$
$\deg(B_1) = (n/2)-1$	$\deg(C_1) = n/2$
$\deg(D) = (3n/4)+1$	$\deg(E) = 3n/4$
$\deg(Q) = 1$	$\deg(F) = (3n/4)-1$

$\deg(G_0) = (n/2)-2$ $\qquad$ $\deg(H_0) = (n/4)+1$

$\deg(G_1) = (n/2)-3$ $\qquad$ $\deg(H_1) = (n/4)+1$

$\deg(M_1) = n/4$ $\qquad$ $\deg(M_2) = (n/4)-1$

It should be noted that the degrees of matrices M_1 and M_2 are assumed to be the same as the degrees of the largest element in each matrix. Under these conditions, the cost of each step in the EMGCD algorithm can be outlined as given below:

Step 3: $n+2$

Step 4: $n+1$

Step 5: $T([n/2]-1)$

Step 7: $4C_M(n/4,n/2)+(9n/2)+7$

Step 9: $C_D([3n/4]+1,3n/4)$

Step 10: $(3n/4)+1$

Step 11: $3n/4$

Step 12: $T([n/2]-3)$

Step 14: $4C_M([n/2]-1,[n/4]+1)+2C_M(1,[n/4]-1)+$
$8C_M(n/4,n/4)+(11n/2)+13$

The remaining steps have no associated cost. After a few minor modifications, this brings the total cost of the EMGCD algorithm to:

$$T(n) = 2T(n/2)+C_D([3n/4]+1,3n/4)+4C_M(n/4,n/2)+12C_M(n/4,n/4)+2C_M(1,[n/4]-1)+(27n/2)+24$$

Using classical polynomial arithmetic, this cost becomes:

$$T(n) = 2T(n/2)+(5n^2/2)+(73n/2)+58$$
$$= 5n^2+(73/2)n\log n+53n-58 \text{ unit operations,}$$

while with fast polynomial arithmetic, this cost is:

$$T(n) = 2T(n/2)+114n\log n+104n+178$$
$$= 57n\log^2 n+161n\log n+178n-178 \text{ unit operations.}$$

Note that this cost is based on the size of the smaller input polynomial. This is not a normal practice. However, when using this algorithm to compute Pade approximants, as outlined shortly, the size of the larger input polynomial is always exactly one more than the size of the smaller one. Therefore, in this case, this convention is valid. It is done because it simplifies the cost comparisons of the anti-diagonal approach for computing Pade approximants with other methods.

Having completed the description of the EMGCD algorithm, it can now be used to compute the reduced (n,n) diagonal Pade approximant for an arbitrary power series, A, in the following way:

Algorithm 3.1.9: AD(P,Q,A,n)

Input: A is a power series,
n is an integer.

Output: P,Q such that P/Q is the reduced (n,n) diagonal Pade approximant for A.

Comment M is a matrix.

1) Call EMGCD(M,x^{2n+1},A mod x^{2n+1}).

2) $\begin{bmatrix} P \\ Q \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} M^T \begin{bmatrix} 0 \\ 1 \end{bmatrix}$.

3) d = min[ord(P),ord(Q)].

4) P=$\lfloor P/x^d \rfloor$, Q=$\lfloor Q/x^d \rfloor$.

The correctness of the first step of this algorithm has already been verified. Step 2 is simply an exotic, but mathematically concise, way of setting the values of P and Q to be the same as specific elements of matrix M. The purpose of the last two steps is to put the rational function P/Q into reduced form. That this particular method for doing so is correct is quaranteed by the following result:

Theorem 3.1.10

Let A_i,T_i be as defined by the extended Euclidean algorithm for all $0 \leq i \leq k$. Then $\gcd(A_i,T_i) \mid A_0$.

This result is a direct consequence of the third property given in Theorem 3.1.2. The implication of this property to the anti-diagonal approach for computing the (m,n) Pade approximant is that any common divisor of any numerator and denominator derived using the extended Euclidean algorithm must also divide x^{m+n+1}. Therefore, any common divisor of such a numerator and denominator, and, thus, of P and Q in Algorithm 3.1.9, must be an integral power of the indeterminate, x.

Determining the cost of Algorithm 3.1.9 is straightforward. The cost of step 1 is the same as the cost of the EMGCD algorithm when $\deg(A_1) = 2n$ while the cost of step 4 is 2n unit operations. Therefore, using classical arithmetic, the cost of the anti-diagonal approach to computing the reduced (n,n) diagonal Pade approximant is:

$$T(n) = 20n^2+73n\log n+181n-58 \text{ unit operations,}$$

whereas using fast polynomial arithmetic, this cost is:

$$T(n) = 114n\log^2 n+550n\log n+794n-178 \text{ unit operations.}$$

3.2 The MD Algorithm

The first of the forward diagonal algorithms is the MD algorithm presented by Brent, Gustavson, and Yun [5]. This algorithm uses another recursive algorithm to compute the (n,n) diagonal Pade approximant for an (n,n)-normal power series quotient. Let A and B be two power series such that ord(A) = ord(B) = 0 and A/B is (n,n)-normal. Let s be an integer with a value of zero or one. Then the reduced (n+s-1,n) and (n+s,n) Pade approximants for A/B can be computed using the following algorithm:

Algorithm 3.2.1: MD2(M,A,B,n,s)

Input: A,B are polynomials such that ord(A)=ord(B)=0 and
A/B is (n,n)-normal,
n is an integer,
s is an integer such that s=0 or s=1.

Output: $M = \begin{bmatrix} P & \bar{P} \\ Q & \bar{Q} \end{bmatrix}$ where P/Q is the reduced $(n+s-1,n)$ Pade approximant for A/B,

and $\bar{P}/\bar{Q}$ is the reduced $(n+s,n)$ Pade approximant for A/B.

Comment C,D,E,F are polynomials,

M_1, M_2 are matrices.

1) If $n+s \le 0$ then $M = \begin{bmatrix} 0 & a_0 \\ 1 & b_0 \end{bmatrix}$ else begin

2) $k = \lfloor n/2 \rfloor$.

3) $d = n-k$.

4) $m = d+s-1$.

5) Call MD2(M_1, B mod x^{n+s+1}, A mod x^{n+s+1}, m, 1-s).

6) $\begin{bmatrix} C \\ D \end{bmatrix} = M_1^T \begin{bmatrix} -A \\ B \end{bmatrix}$.

7) $E = \lfloor C/x^{m+d} \rfloor \bmod x^{n+1}$, $F = \lfloor D/x^{m+d+1} \rfloor \bmod x^{n+1}$.

8) Call MD2(M_2, F, E, k, 0).

9) $\begin{bmatrix} Q & \bar{Q} \\ P & \bar{P} \end{bmatrix} = M_1 \begin{bmatrix} -x & 0 \\ 0 & 1 \end{bmatrix} M_2$.

10) $M = \begin{bmatrix} P \bmod x^{n+s} & \bar{P} \bmod x^{n+s+1} \\ Q \bmod x^{n+1} & \bar{Q} \bmod x^{n+1} \end{bmatrix}$.

end.

The proof of this algorithm's validity is, again, by induction.

Let $P_{i,j}(\cdot)$ denote the (i,j) Pade approximant of a power series.

Let $M_1 = \begin{bmatrix} Q_1 & \bar{Q}_1 \\ P_1 & \bar{P}_1 \end{bmatrix}$ and $M_2 = \begin{bmatrix} Q_2 & \bar{Q}_2 \\ P_2 & \bar{P}_2 \end{bmatrix}$.

Assume that M_1 and M_2 are correct.

Then $Q_1/P_1 = P_{d-1,m}(B/A) \quad \Rightarrow \quad P_1/Q_1 = P_{m,d-1}(A/B)$,

$\bar{Q}_1/\bar{P}_1 = P_{d,m}(B/A) \quad \Rightarrow \quad \bar{P}_1/\bar{Q}_1 = P_{m,d}(A/B)$,

$Q_2/P_2 = P_{k-1,k}(F/E) \quad \Rightarrow \quad P_2/Q_2 = P_{k,k-1}(E/F)$,

$\bar{Q}_2/\bar{P}_2 = P_{k,k}(F/E) \quad \Rightarrow \quad \bar{P}_2/\bar{Q}_2 = P_{k,k}(E/F)$.

Furthermore, $ord(P_i) = ord(\bar{P}_i) = ord(Q_i) = ord(\bar{Q}_i) = 0$ for $i=1,2$.

Because of normality, $ord(E) = ord(F) = 0$.

Thus, $PB-QA = (\bar{P}_1P_2-xP_1Q_2)B-(\bar{Q}_1P_2-xQ_1Q_2)A$

$= (\bar{P}_1B-\bar{Q}_1A)P_2-x(P_1B-Q_1A)Q_2$

$= (FP_2-EQ_2)x^{m+d+1}$

$= 0 \mod x^{2n+s-1}$.

Also, $deg(P) = max[deg(\bar{P}_1)+deg(P_2), deg(P_1)+deg(Q_2)+1]$

$= n+s-1$,

$deg(Q) = max[deg(\bar{Q}_1)+deg(P_2), deg(Q_1)+deg(Q_2)+1]$

$= n$,

$ord(P) = ord(\bar{P}_1P_2) = 0$ since $ord(xP_1Q_2) = 1$,

$ord(Q) = ord(\bar{Q}_1P_2) = 0$ since $ord(xQ_1Q_2) = 1$.

Therefore, P/Q is the reduced $(n+s-1,n)$ Pade approximant for A/B.

By an identical argument, $\bar{P}/\bar{Q}$ is the reduced $(n+s,n)$ Pade approximant for A/B.

Thus, the algorithm behaves as stated.

For determining the cost of the MD2 algorithm, a few conventions are again adopted. In particular, it is again

assumed that n is an exact power of 2. Furthermore, it is assumed that s = 1. The effect of these two assumptions is that k = d = m = n/2. Under these conditions, and remembering that A/B is (n,n)-normal, the sizes of various intermediate results can be given as follows:

$\deg(A) = 2n+1$ $\qquad$ $\deg(C) = [5n/2]+1$

$\deg(B) = 2n+1$ $\qquad$ $\deg(D) = [5n/2]+1$

$\deg(M_1) = n/2$ $\qquad$ $\deg(M_2) = n/2$

In fact, matrices M_1 and M_2 each contain one polynomial of size [n/2]-1 and three of size n/2. However, setting the size of all to n/2 should only have a very minor effect, for large n, on the final equation. Based on these values, the cost of each step in the algorithm can now be summarized as below:

Step 5: $T(n/2)$

Step 6: $4C_M(n/2,2n+1)+5n+4$

Step 7: $8n+7$

Step 8: $T(n/2)$

Step 9: $4C_M(n/2,[n/2]+1)+4C_M(n/2,n/2)+5n+12$

Step 10: $4n+8$

The remaining steps are cost-free. Thus, the total cost of this algorithm becomes:

$$T(n) = 2T(n/2)+4C_M(n/2,2n+1)+4C_M(n/2,[n/2]+1)+4C_M(n/2,n/2)+22n+31.$$

Using classical polynomial arithmetic, this cost becomes:

$$T(n) = 2T(n/2)+12n^2+66n+71$$
$$= 24n^2+66n\log n+47n-71 \text{ unit operations,}$$

whereas using fast arithmetic, it becomes:

$$T(n) = 2T(n/2)+192n\log n+470n+31$$
$$= 96n\log^2 n+566n\log n+31n-31 \text{ unit operations.}$$

It is interesting to note that no polynomial or power series division is required by this algorithm. This accounts for its good performance when using fast arithmetic. To offset this, however, is the fact that A/B must be (n,n)-normal. Brent, Gustavson, and Yun [5] mention that this normality requirement can be relaxed somewhat. However, then the resulting rational function would not necessarily be in reduced form anymore. Furthermore, they indicate that to remove the normality condition entirely deteriorates the complexity of the algorithm. They do not give any further information about either claim so these points are not pursued any further.

To compute the reduced (n,n) diagonal Pade approximant for any (n,n)-normal power series using this technique, all that is now required is an appropriate call to the MD2 procedure. If A is an (n,n)-normal power series and n is an integer, this is accomplished as follows:

Algorithm 3.2.2: MD(P,Q,A,n)

Input: A is an (n,n)-normal power series,
n is an integer.

Output: P,Q such that P/Q is the reduced (n,n) Pade approximant for A.

Comment M is a matrix.

1) Call MD2(M,A,1,n,0).

2) $\begin{bmatrix} P \\ Q \end{bmatrix} = M \begin{bmatrix} 0 \\ 1 \end{bmatrix}$.

The validity of this algorithm is directly dependent on the correctness of the MD2 algorithm. The cost of the MD algorithm is identical to the cost of MD2.

3.3 The DIAG Algorithm

The next forward diagonal approach for computing diagonal Pade approximants is an iterative method developed by Cabay and Kao [6]. Given two consecutive diagonal Pade approximants, this technique computes the next one in the sequence based on the given two. Let A be an arbitrary power series truncated to 2n+1 terms. Then this approach computes the reduced (n,n) diagonal Pade approximant by the following algorithm:

Algorithm 3.3.1: DIAG(P,Q,A,n)

Input: A is a power series truncated to 2n+1 terms,

n is an integer.

Output: P,Q such that P/Q is the reduced (n,n) Pade approximant for A.

Comment B, P_k, Q_k, R_k are power series for all k.

1) $\begin{bmatrix} P_0 & P_{-1} \\ Q_0 & Q_{-1} \end{bmatrix} = \begin{bmatrix} a_0 & -x^{-1} \\ 1 & 0 \end{bmatrix}$.

2) $R_{-1}=1$, $k=0$, $j=-1$, $i=0$.

3) While $i < n$ do begin

4) $\quad d = \text{ord}[\lfloor (AQ_k)/x^{2i+1} \rfloor]+i+1$.

5) $\quad$ If $d \le n$ then begin

6) $\quad\quad R_k = \lfloor (AQ_k)/x^{d+i} \rfloor \bmod x^{d-i+1}$.

7) $\quad\quad R_{k-1} = \lfloor (AQ_{k-1})/x^{i+j} \rfloor \bmod x^{d-i+1}$.

8) $\quad\quad B = (R_{k-1}/R_k) \bmod x^{d-i+1}$.

9) $\quad\quad \begin{bmatrix} P_{k+1} \\ Q_{k+1} \end{bmatrix} = \begin{bmatrix} P_k & P_{k-1} \\ Q_k & Q_{k-1} \end{bmatrix} \begin{bmatrix} B \\ -x^{d-j} \end{bmatrix}$.

10) $\quad\quad k = k+1$.

$\quad$ end.

11) $\quad j = i$.

12) $\quad i = d$.

end.

13) $P=P_k$, $Q=Q_k$.

The division in step 8 must be done using power series division. The division by the power of x in steps 4, 6, and 7 must be done using polynomial division. All other calculations involving power series can be done using either power series or polynomial arithmetic. In the case of this

algorithm, polynomial arithmetic is more efficient. The proof of this algorithm is by induction:

Assume that P_k/Q_k is the reduced (i,i) Pade approximant for A and that P_{k-1}/Q_{k-1} is the reduced (j,j) Pade approximant for A, with i > j.

Assume also that $ord(Q_k) = ord(Q_{k-1}) = 0$.

Then $AQ_k-P_k = \lfloor (AQ_k)/x^{2i+1} \rfloor x^{2i+1} + [(AQ_k-P_k) \bmod x^{2i+1}]$

$= \lfloor (AQ_k)/x^{2i+1} \rfloor x^{2i+1}$.

Thus, $ord(AQ_k-P_k) = d+i \Rightarrow AQ_k-P_k = 0 \bmod x^{d+i}$.

Therefore,

$$AQ_k-P_k = \lfloor (AQ_k)/x^{d+i} \rfloor x^{d+i} + [(AQ_k-P_k) \bmod x^{d+i}]$$
$$= (R_k \bmod x^{d-i+1}) x^{d+i}$$
$$= (R_k x^{d+i}) \bmod x^{2d+1}.$$

Furthermore, $ord(R_k) = ord(AQ_k-P_k)-d-i = 0$.

Similarly, $ord(AQ_{k-1}-P_{k-1}) = i+j$, $ord(R_{k-1}) = 0$, and

$AQ_{k-1}-P_{k-1} = (R_{k-1}x^{i+j}) \bmod x^{d+j+1}$.

Because $ord(R_{k-1}) = ord(R_k) = 0$, $ord(B) = 0$.

The inductive step of the proof proceeds as follows:

$$AQ_{k+1}-P_{k+1} = A(Q_kB-Q_{k-1}x^{d-j})-(P_kB-P_{k-1}x^{d-j})$$
$$= B(AQ_k-P_k)-(AQ_{k-1}-P_{k-1})x^{d-j}$$
$$= [B(R_kx^{d+i})-(R_{k-1}x^{i+j})x^{d-j}] \bmod x^{2d+1}$$
$$= [(BR_k-R_{k-1})x^{d+i}] \bmod x^{2d+1}$$
$$= [(0 \bmod x^{d-i+1})x^{d+i}] \bmod x^{2d+1}$$
$$= 0 \bmod x^{2d+1}.$$

Also, $deg(P_{k+1}) = max[deg(P_k)+deg(B), deg(P_{k-1})+d-j]$

$\leq d$,

$$\deg(Q_{k+1}) = \max[\deg(Q_k)+\deg(B), \deg(Q_{k-1})+d-j]$$
$$\leq d,$$
$$\operatorname{ord}(Q_{k+1}) = \operatorname{ord}(Q_k B) = 0$$
since $\operatorname{ord}(Q_{k-1}x^{d-j}) = d-j$.

Now, observe that $P_{k+1}Q_k - Q_{k+1}P_k = (-1)^{k+1}x^{d+i}$.

This can be verified by induction:

$$\begin{aligned} P_{k+1}Q_k - Q_{k+1}P_k &= (P_k B - P_{k-1}x^{d-j})Q_k - (Q_k B - Q_{k-1}x^{d-j})P_k \\ &= -x^{d-j}(P_k Q_{k-1} - Q_k P_{k-1}) \\ &= (-1)x^{d-j}(-1)^k x^{i+j} \\ &= (-1)^{k+1}x^{d+i}. \end{aligned}$$

This means that $\gcd(P_{k+1}, Q_{k+1}) \mid x^{d+i}$.

However, because $\operatorname{ord}(Q_{k+1}) = 0$, $x \nmid \gcd(P_{k+1}, Q_{k+1})$.

Therefore, $\gcd(P_{k+1}, Q_{k+1}) = 1$.

Consequently, P_{k+1}/Q_{k+1} is the reduced (d,d) Pade approximant for A.

The algorithm terminates when $d \geq n+1$. That this actually occurs is guaranteed by the fact that $d > i$.

At that point, P_k/Q_k forms the reduced (n,n) Pade approximant for A.

Therefore, as calculated by this algorithm, P/Q is the reduced (n,n) Pade approximant for A.

Before performing the cost analysis for the DIAG algorithm, a few comments should be made regarding some of the calculations performed in this algorithm. Consider, first, the problem of computing the value for d in step 4 of the algorithm. One way to obtain the value of

ord[$\lfloor (AQ_k)/x^{2i+1} \rfloor$] is to actually compute the value of AQ_k to 2n terms and then look for the first non-zero term past the (2i)th coefficient of the result. Another approach is to only compute coefficients $2i+1,\ldots,d+i$ of AQ_k until the (d+i)th coefficient is non-zero or until $d+i > 2n$. Using this latter scheme, step 4 of DIAG can be replaced with the following sequence of calculations:

4a) c=0, d=2i.
4b) While c = 0 and d < 2n do begin
4c) d = d+1.
4d) For m = 0 until i do begin
4e) $c = c+a_{d-m}q_{k,m}$.
  end.
end.
4f) If c = 0 then d = n+1 else d = d-i.

where $q_{k,m}$ denotes the mth coefficient of Q_k. In a similar fashion, the value of R_k, computed in step 6 of DIAG, can be obtained by constructing only coefficients $d+i,\ldots,2d$ of the product AQ_k, which can be done by the following method:

6a) For m = d+i until 2d do begin
6b) $r_{k,m} = 0$.
6c) For s = 0 until i do begin
6d) $r_{k,m} = r_{k,m}+a_{m-s}q_{k,s}$.
  end.
end.

where $r_{k,m}$ is the mth coefficient of R_k and $q_{k,s}$ is the sth coefficient of Q_k. Note that if the alternate technique for step 4 is also used, then $r_{k,d+i} = c$ where c is the last coefficient calculated by the first technique and the value for m in step 6a, above, starts at d+i+1 rather than d+i.

At the beginning of each execution of the loop starting at step 5 of the DIAG algorithm, R_{k-1} contains the value $R_{k-1} = \lfloor (AQ_{k-1})/x^{i+j} \rfloor \bmod x^{2i+1}$, as left over from the previous execution of this loop. Thus, at step 7 of the algorithm, it is only necessary to perform any calculations if $2i < d+j$, in which case only coefficients $2i+1,\ldots,d+j$ of the product AQ_{k-1} are required. Consequently, step 7 of this algorithm can be replaced with:

```
7a) If d+j > 2i then begin
7b)     For m = 2i+1 until d+j do begin
7c)         r(k-1,m) = 0.
7d)         For s = 0 until j do begin
7e)             r(k-1,m) = r(k-1,m) + a(m-s) q(k-1,s).
            end.
        end.
    end.
```

where $r_{k-1,m}$ is the mth coefficient of R_{k-1} and $q_{k-1,s}$ is the sth coefficient of Q_{k-1}.

The three techniques presented above are only valuable if they represent an improvement over the original methods for producing the same results. A careful examination of

these techniques shows that each computes a partial power series (or polynomial) product using classical power series multiplication. Thus, each is an immediate improvement, by necessity, over computing the complete product using classical arithmetic. That they are an improvement over computing the total product using fast power series multiplication results from the fact that, in the worst case, steps 4, 6, and 7 of DIAG are each executed n times. It is shown shortly that using these three techniques then results in a total algorithm complexity of $O(n^2)$. On the other hand, computing the total product using fast power series arithmetic results in an $O(n^2 \log n)$ algorithm. Therefore, computing only the partial products in this way represents a clear improvement over computing the total product using either form of power series arithmetic. It should be noted that a partial product can not be obtained using the fast power series multiplication technique without first computing the total product. Thus, no further gain can be obtained in that direction.

For determining the cost of the DIAG algorithm, assume that the worst case situation exists, which again occurs when the input power series, A, is normal along the main diagonal. In that event, the following bounds hold on the sizes of various calculated values:

$\deg(P_k) = k$ $\qquad$ $\deg(R_{k-1}) = 1$

$\deg(Q_k) = k$ $\qquad$ $\deg(R_k) = 1$

$\deg(B) = 1$

Then, assuming that the three improvements presented above are used, the cost of each non-trivial step of DIAG, excluding the loop starting at step 5, can be given as follows:

Step 4: $2k+2$

Step 6: $2k+2$

Step 7: 0

Step 8: $C_D(1,1)$

Step 9: $2C_M(k,1)+4k+8$

This brings the total cost of the algorithm, as determined by the loop starting at step 5, to:

$$T(n) = \sum_{k=0}^{n-1} [2C_M(k,1)+C_D(1,1)+8k+12].$$

Using classical polynomial multiplication and power series division, this equation collapses to:

$$T(n) = \sum_{k=0}^{n-1} (12k+20) = 6n^2+14n \text{ unit operations.}$$

Cabay and Kao [6] derive the same bound for classical arithmetic usage without the normality assumption. Using the fast arithmetic algorithms, and assuming that n is an exact power of 2, the total cost becomes:

$$T(n) = \sum_{k=0}^{n-1} (12m\log m+10m+8k+162)$$

$$= 8n^2\log n+8n^2+48n\log n+246n-48 \text{ unit operations}$$

where $m = 2^{\lceil \log(k+2) \rceil}$. It is interesting to note that for this particular algorithm, use of the fast arithmetic

techniques actually results in a poorer complexity for the worst case situation than use of the classical methods.

3.4 The OFFDIAG Algorithm

The final algorithm considered for computing symbolic diagonal Pade approximants was developed by Choi [7] in an attempt to improve on the complexity of the technique used in the DIAG algorithm. The result is an algorithm with a better asymptotic cost which can be used to compute the Pade approximants of a power series on or below the main diagonal of the Pade table. Let A be a power series truncated to m+n+1 terms where m and n are two integers such that $m \geq n$. Then the OFFDIAG algorithm produces two Pade approximants for A along the diagonal of the Pade table for which $i-j = m-n$. One is the reduced (m,n) Pade approximant for A. The other is the last different Pade approximant for A to appear on the same diagonal just before the (m,n) Pade approximant.

Algorithm 3.4.1: OFFDIAG(M,A,m,n,$\bar{m}$,$\bar{n}$)

Input: A is a power series truncated to m+n+1 terms,
m,n are integers such that $m \geq n$.

Output: $M = \begin{bmatrix} P & \bar{P} \\ Q & \bar{Q} \end{bmatrix}$ where P/Q is the reduced (m,n) Pade approximant for A,
and $\bar{P}/\bar{Q}$ is the $(\bar{m}-1,\bar{n}-1)$ Pade approximant for A,
$\bar{m},\bar{n}$ such that $\bar{m}+\bar{n}=\text{ord}(AQ-P)-1$ and $\bar{m}-\bar{n}=m-n$.

Comment $A^*, R, \bar{R}$ are power series,
M^* is a matrix.

1) $d=m-n$, $i=0$.

2) $\bar{m}=d$, $\bar{n}=0$.

3) $\begin{bmatrix} P & \bar{P} \\ Q & \bar{Q} \end{bmatrix} = \begin{bmatrix} (A \bmod x^{d+1}) & -x^{d-1} \\ 1 & 0 \end{bmatrix}$.

4) While $2^i < 2n$ do begin

5) If $n > 2^i$ then $m^* = 2^i - \bar{n}$ else $m^* = n - \bar{n}$.

6) $R = \lfloor (AQ)/x^{\bar{m}+\bar{n}+1} \rfloor \bmod x^{2m^*}$.

7) If $R \neq 0$ then begin

8) $d^* = \mathrm{ord}(R)+1$.

9) $n^* = m^* - d^*$.

10) $R = \lfloor R/x^{d^*-1} \rfloor$.

11) $\bar{R} = \lfloor (A\bar{Q})/x^{\bar{m}+\bar{n}-1} \rfloor \bmod x^{m^*+n^*+1}$.

12) $A^* = (\bar{R}/R) \bmod x^{m^*+n^*+1}$.

13) Call OFFDIAG($M^*, A^*, m^*, n^*, \bar{m}^*, \bar{n}^*$).

14) $\begin{bmatrix} P & \bar{P} \\ Q & \bar{Q} \end{bmatrix} = \begin{bmatrix} P & \bar{P} \\ Q & \bar{Q} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & -x^{d^*+1} \end{bmatrix} M^*$.

15) $\bar{m}=\bar{m}+\bar{m}^*$, $\bar{n}=\bar{n}+\bar{m}^*$.

end.

16) $i = i+1$.

end.

17) $M = \begin{bmatrix} P & \bar{P} \\ Q & \bar{Q} \end{bmatrix}$.

Note that the Pade approximant computed as $\bar{P}/\bar{Q}$ is not necessarily in reduced form. As with the DIAG algorithm, the

division in step 12 of OFFDIAG must be done using power series division while the division by the power of x in steps 6, 10, and 11 must by done using polynomial division. All other power series operations can be done using either power series or polynomial arithmetic, although for step 14, the polynomial methods are more efficient. The proof of the OFFDIAG algorithm is very similar to that for DIAG:

For notational convenience, replace step 14 with

$$\begin{bmatrix} P' & \bar{P}' \\ Q' & \bar{Q}' \end{bmatrix} = \begin{bmatrix} P & \bar{P} \\ Q & \bar{Q} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & -x^{\bar{d}^*+1} \end{bmatrix} M^*.$$

Let $M^* = \begin{bmatrix} P^* & \bar{P}^* \\ Q^* & \bar{Q}^* \end{bmatrix}$.

Assume that $\begin{bmatrix} P & \bar{P} \\ Q & \bar{Q} \end{bmatrix}$ and M^* are correct, forming the inductive hypothesis.

Then $AQ-P = 0 \bmod x^{\bar{m}+\bar{n}+1}$, $\quad AQ^*-P^* = 0 \bmod x^{\bar{m}^*+\bar{n}^*+1}$,

$A\bar{Q}-\bar{P} = 0 \bmod x^{\bar{m}+\bar{n}-1}$, $\quad A\bar{Q}^*-\bar{P}^* = 0 \bmod x^{\bar{m}^*+\bar{n}^*-1}$.

Also, $\mathrm{ord}(Q) = \mathrm{ord}(Q^*) = 0$ since both P/Q and P^*/Q^* are in reduced form.

This means that

$$AQ-P = \lfloor (AQ)/x^{\bar{m}+\bar{n}+1} \rfloor x^{\bar{m}+\bar{n}+1} + [(AQ-P) \bmod x^{\bar{m}+\bar{n}+1}]$$
$$= \lfloor (AQ)/x^{\bar{m}+\bar{n}+1} \rfloor x^{\bar{m}+\bar{n}+1}.$$

Thus $\mathrm{ord}(AQ-P) = \mathrm{ord}[\lfloor (AQ)/x^{\bar{m}+\bar{n}+1} \rfloor] + \bar{m}+\bar{n}+1 = \bar{m}+\bar{n}+d^*$,

$\Rightarrow \quad AQ-P = 0 \bmod x^{\bar{m}+\bar{n}+d^*}$.

Therefore,

$$AQ-P = \lfloor (AQ)/x^{\bar{m}+\bar{n}+d^*} \rfloor x^{\bar{m}+\bar{n}+d^*} + [(AQ-P) \bmod x^{\bar{m}+\bar{n}+d^*}]$$
$$= (R \bmod x^{2m^*-d^*+1}) x^{\bar{m}+\bar{n}+d^*}$$

$$= (Rx^{\bar{m}+\bar{n}+d^*}) \bmod x^{2m^*+\bar{m}+\bar{n}+1}.$$

Furthermore, $ord(R) = ord(AQ-P)-\bar{m}-\bar{n}-d^* = 0$.

Similarly, $ord(A\bar{Q}-\bar{P}) = \bar{m}+\bar{n}-1$, $ord(\bar{R}) = 0$, and

$$A\bar{Q}-\bar{P} = (\bar{R}x^{\bar{m}+\bar{n}-1}) \bmod x^{m^*+n^*+\bar{m}+\bar{n}}.$$

Because $ord(R) = ord(\bar{R}) = 0$, $ord(A^*) = 0$.

The inductive step of the proof proceeds as follows:

$$\begin{aligned}
AQ'-P' &= A(QP^*-\bar{Q}x^{d^*+1}Q^*)-(PP^*-\bar{P}x^{d^*+1}Q^*) \\
&= (AQ-P)P^*-(A\bar{Q}-\bar{P})x^{d^*+1}Q^* \\
&= [(Rx^{\bar{m}+\bar{n}+d^*}) \bmod x^{2m^*+\bar{m}+\bar{n}+1}]P^*- \\
&\quad [(\bar{R}x^{\bar{m}+\bar{n}-1}) \bmod x^{m^*+n^*+\bar{m}+\bar{n}}]x^{d^*+1}Q^* \\
&= [(Rx^{\bar{m}+\bar{n}+d^*}) \bmod x^{2m^*+\bar{m}+\bar{n}+1}]P^*- \\
&\quad [(\bar{R}x^{\bar{m}+\bar{n}+d^*}) \bmod x^{2m^*+\bar{m}+\bar{n}+1}]Q^* \\
&= [(RP^*-\bar{R}Q^*)x^{\bar{m}+\bar{n}+d^*}] \bmod x^{2m^*+\bar{m}+\bar{n}+1}.
\end{aligned}$$

$$\begin{aligned}
\Rightarrow R^{-1}(AQ'-P') &= [(P^*-R^{-1}\bar{R}Q^*)x^{\bar{m}+\bar{n}+d^*}] \bmod x^{2m^*+\bar{m}+\bar{n}+1} \\
&= -[(A^*Q^*-P^*)x^{\bar{m}+\bar{n}+d^*}] \bmod x^{2m^*+\bar{m}+\bar{n}+1} \\
&= 0 \bmod x^{\bar{m}+\bar{n}+d^*+\bar{m}^*+\bar{n}^*+1} \\
&= 0 \bmod x^{(\bar{m}+\bar{m}^*)+(\bar{n}+\bar{m}^*)+1}.
\end{aligned}$$

Since $R^{-1} \neq 0$, this implies that

$$AQ'-P' = 0 \bmod x^{(\bar{m}+\bar{m}^*)+(\bar{n}+\bar{m}^*)+1}.$$

$$\begin{aligned}
\text{Also, } deg(P') &= \max[deg(PP^*), deg(\bar{P}Q^*)+d^*+1] \\
&\leq \bar{m}+\bar{m}^*, \\
deg(Q') &= \max[deg(QP^*), deg(\bar{Q}Q^*)+d^*+1] \\
&\leq \bar{n}+\bar{m}^*.
\end{aligned}$$

Thus, P'/Q' is the $(\bar{m}+\bar{m}^*, \bar{n}+\bar{m}^*)$ Pade approximant for A.

By a similar argument, $\bar{P}'/\bar{Q}'$ forms the $(\bar{m}+\bar{m}^*-1, \bar{n}+\bar{m}^*-1)$ Pade approximant for A.

By induction, $ord(P') = \min[ord(A), ord(\bar{P}Q^*)+d^*+1]$, as

shown by:

$$\begin{aligned}\mathrm{ord}(P') &= \min[\mathrm{ord}(PP^*),\mathrm{ord}(\bar{P}Q^*)+d^*+1]\\ &= \min[\mathrm{ord}(P)+\mathrm{ord}(P^*),\mathrm{ord}(\bar{P}Q^*)+d^*+1]\\ &= \min[\mathrm{ord}(A)+\mathrm{ord}(A^*),\mathrm{ord}(\bar{P}Q^*)+d^*+1]\\ &= \min[\mathrm{ord}(A),\mathrm{ord}(\bar{P}Q^*)+d^*+1].\end{aligned}$$

Thus, in particular, $\mathrm{ord}(P^*) = 0$ since $\mathrm{ord}(A^*) = 0$.

Then $\mathrm{ord}(Q') = \min[\mathrm{ord}(QP^*),\mathrm{ord}(\bar{Q}Q^*)+d^*+1] = 0$.

Now observe that $P'\bar{Q}'-Q'\bar{P}' = -x^k$ for some $k \geq 0$.

Assuming that $P\bar{Q}-Q\bar{P} = -x^h$ and $P^*\bar{Q}^*-Q^*\bar{P}^* = -x^j$ for some h and j, this can be verified by induction:

$$\begin{aligned}P'\bar{Q}'-Q'\bar{P}' &= [(PP^*-x^{d^*+1}Q^*\bar{P})(Q\bar{P}^*-x^{d^*+1}\bar{Q}^*\bar{Q})]-\\ &\quad [(QP^*-x^{d^*+1}Q^*\bar{Q})(P\bar{P}^*-x^{d^*+1}\bar{Q}^*\bar{P})]\\ &= -x^{d^*+1}[Q^*\bar{P}Q\bar{P}^*+PP^*\bar{Q}^*\bar{Q}-Q^*\bar{Q}P\bar{P}^*-QP^*\bar{Q}^*\bar{P}]\\ &= -x^{d^*+1}[(P\bar{Q}P^*\bar{Q}^*-Q\bar{P}P^*\bar{Q}^*)+\\ &\qquad (Q\bar{P}Q^*\bar{P}^*-P\bar{Q}Q^*\bar{P}^*)]\\ &= -x^{d^*+1}[(P\bar{Q}P^*\bar{Q}^*-Q\bar{P}P^*\bar{Q}^*)-\\ &\qquad (P\bar{Q}Q^*\bar{P}^*-Q\bar{P}Q^*\bar{P}^*)]\\ &= -x^{d^*+1}(P\bar{Q}-Q\bar{P})(P^*\bar{Q}^*-Q^*\bar{P}^*)\\ &= (-x^{d^*+1})(-x^h)(-x^j)\\ &= -x^k\end{aligned}$$

where $k = h+j+d^*+1$.

This means that $\gcd(P',Q') \mid x^k$.

However, because $\mathrm{ord}(Q') = 0$, $x \nmid \gcd(P',Q')$.

Therefore, $\gcd(P',Q') = 1$.

Consequently, P'/Q' is in reduced form.

It is now a trivial task to verify that at some point, P'/Q' forms the (m,n) Pade approximant for A.

The details of the fact that this occurs when $i \geq \log n$ are left to Choi [7].

This then completes the verification for this algorithm.

Like in the DIAG algorithm, the calculations performed in steps 6, 10, and 11 can be performed either as stated or by computing only a partial power series product using the classical power series multiplication technique. However, in this case, use of the second alternative results in a poorer complexity when fast power series multiplication is used. This occurs because of the fact that the covering loop, starting at step 4 of OFFDIAG, is only executed log n times whereas in DIAG, it is executed n times in the worst case situation. Consequently, use of the partial multiplication scheme is not considered for the OFFDIAG algorithm.

Assume now that the input power series, A, is (m,n)-normal, which represents the worst-case situation for the OFFDIAG algorithm. In that case, the following assertions hold about the sizes of various values before step 14 of the algorithm is executed:

$$\deg(P) = \bar{m} \qquad \deg(P^*) = m^*$$
$$\deg(Q) = \bar{n} \qquad \deg(Q^*) = n^*$$
$$\deg(\bar{P}) = \bar{m}-1 \qquad \deg(\bar{P}^*) = m^*-1$$
$$\deg(\bar{Q}) = \bar{n}-1 \qquad \deg(\bar{Q}^*) = n^*-1$$
$$\deg(R) = m^*+n^* \qquad \deg(A^*) = m^*+n^*$$
$$\deg(\bar{R}) = m^*+n^*$$

This means that the costs of the non-trivial steps in the

algorithm can be summarized as follows:

Step 6: $C_M(2m^*+\bar{m}+\bar{n},\bar{n})+4m^*+2\bar{m}+3\bar{n}+2$

Step 10: $2m^*$

Step 11: $C_M(m^*+n^*+\bar{m}+\bar{n}-1,\bar{n})+2m^*+2n^*+2\bar{m}+3\bar{n}$

Step 12: $C_D(m^*+n^*,m^*+n^*)$

Step 13: $T(m^*,n^*)$

Step 14: $4C_M(\bar{m},m^*)+4C_M(\bar{n},m^*)+8m^*+2\bar{m}+2\bar{n}+8$

Since the major concern of this discussion is diagonal Pade approximants, let $m = n$, which then implies that $\bar{m} = \bar{n}$. Assume further that n is an exact power of 2. Then, when $i = 0$, $\bar{m} = \bar{n} = n^* = 0$ and $m^* = 1$, while if $i \geq 1$, $\bar{m} = \bar{n} = m^* = 2^{i-1}$ and $n^* = 2^{i-1}-1$. This means, after a couple of minor modifications, that

$$T(1) = C_D(1,1)+2C_M(2,0)+8C_M(0,1)+26, \quad \text{and}$$

$$T(2^i) = T(2^{i-1})+C_D(2^i-1,2^i-1)+2C_M(2^{i+1},2^{i-1})+ 8C_M(2^{i-1},2^{i-1})+16(2^i)+10$$

where $i > 0$. This brings the total cost of the OFFDIAG algorithm to:

$$T(n) = 2T(n/2)+C_D(n-1,n-1)+2C_M(2n,n/2)+8C_M(n/2,n/2)+ 16n+C_D(1,1)+2C_M(2,0)+8C_M(0,1)+36.$$

Using classical arithmetic techniques, this cost becomes:

$$T(n) = 2T(n/2)+9n^2+42n+104$$
$$= 18n^2+42n\log n+160n-104 \text{ unit operations.}$$

The fast arithmetic methods produce a total cost of:

$$T(n) = 2T(n/2)+186n\log n+361n+498$$
$$= 93n\log^2 n+454n\log n+986n-498 \text{ unit operations.}$$

To use the technique of the OFFDIAG algorithm for computing the reduced (n,n) diagonal Pade approximant for an arbitrary power series, A, it is now only necessary to invoke the following procedure:

Algorithm 3.4.2: OFFDIAGnn(P,Q,A,n)

Input: A is an arbitrary power series,
n is an integer.

Output: P,Q such that P/Q is the reduced (n,n) Pade approximant for A.

Comment: M is a matrix,
i,j are integers.

1) Call OFFDIAG(M,A,n,n,i,j).

2) $\begin{bmatrix} P \\ Q \end{bmatrix} = M \begin{bmatrix} 1 \\ 0 \end{bmatrix}$.

The validity of this algorithm is guaranteed by the correctness of OFFDIAG. The cost of this procedure is identical to the cost of the OFFDIAG algorithm.

4. Practical Analysis

In order to properly present an empirical analysis of the diagonal Pade algorithms, it is necessary to consider three practical aspects pertaining to empirical timing results. To begin with, some attention must be paid to the selection of the actual field used as the polynomial and power series coefficient domain. Next, some specifics of the implementation must be included. Finally, and most importantly, the timing results, and how they were obtained, must be presented and discussed.

4.1 Selection of the Coefficient Field

Up to this point, it has been assumed that the coefficients of all polynomials and power series lie in an arbitrary field. For a practical implementation, it is necessary to select a specific field as the coefficient domain. One of the more popular classes of fields used for this purpose are the finite fields, or Galois fields, consisting of the integers modulo a prime integer. For any given prime integer, p, such a field is often denoted by GF(p). The presentation of the nature and properties of such fields are outside the scope of the current discussion. Details can be found in any introductory text on number theory, for example, in Shockley [25].

Although adequate for many algebraic applications, the use of GF(p) for any arbitrary prime integer, p, is not completely sufficient for the problem at hand. The reason

for this is that the fast polynomial and power series arithmetic techniques require the use of nth roots of unity, for specific values of n. Verifying the existence of roots of unity, and finding them if they do exist, is a non-trivial task. Some assistance is provided by the following result:

Theorem 4.1.1

Let p be a prime integer. Then GF(p) has a primitive nth root of unity if $n \mid (p-1)$.

The proof of this theorem can be found in Shockley [25]. Using this result in conjunction with the fact that n always has the form $n = 2^k$ for some $k > 0$, it becomes immediately apparent that the optimal choice for GF(p) is where p is a Fourier prime.

Definition 4.1.2

A prime integer, p, is a Fourier prime if $p = c2^m+1$ for some $c,m > 0$.

Some valuable work regarding the existence and practical use of Fourier primes has been done by J. Lipson. Details of his results can be found in Lipson [19]. The actual Fourier prime used in the present implementation will be given shortly.

One item that has been largely ignored to this point is the problem of how to compute inverses within the

coefficient field. The major reason for this delay is that such calculations tend to be very dependent on the domain within which they are done. In GF(p), multiplicative inverses can be computed using a couple of different techniques. The better of the two, as determined in an analysis by Collins [8], is the extended Euclidean algorithm. Since gcd(b,p) = 1 for any non-zero value b in GF(p), the extended Euclidean algorithm can be used to compute, using standard integer division, values c and d in GF(p) such that

$$cb + dp = 1 \quad \Rightarrow \quad cb = 1 \bmod p.$$

Consequently, c is the multiplicative inverse of b in GF(p). A result attributed to G. Lame, as given by Horowitz and Sahni [14], states that the number of divisions required by the extended Euclidean algorithm in this case is no more than $5\log_{10}b$. This means that for any given p, the number of operations required to compute the inverse of a non-zero element in GF(p) is bounded by a constant.

4.2 Implementation Details

The algorithms described in the previous chapters have been implemented in an extended form of the AlgolW programming language using a slightly modified version of a compiler jointly developed by the University of Newcastle upon Tyne and the University of Michigan. A description of both the language and the compiler used can be found in

[24]. The implementation was accomplished on the MTS operating system, a somewhat dated but still fairly accurate general description of which can be found in the article by Boettner and Alexander [3].

One of the features of the AlgolW language is that any integer product cannot have a result larger than 2147483647. Thus, if exact arithmetic is to be retained, no element of the GF(p) used as the coefficient field can be larger than 46340. Since 40961 is the largest Fourier prime within this bound, GF(40961) becomes the optimal choice for use as the coefficient domain in AlgolW, and is the field so used in the present implementation.

The largest value of n, where n is a power of 2, such that an nth primitive root of unity exists in GF(40961) is n = 4096, which means that the fast polynomial and power series multiplication technique can be used to obtain products with a maximum degree of 4095. An empirical examination shows that 3 is an appropriate (4096)th primitive root of unity for this field. All other primitive roots of unity required for obtaining smaller results using fast arithmetic are simple integral powers of this given primitive root. To cut down on the affect of manipulating roots of unity on the execution times obtained, all powers of this primitive root are stored, in the present implementation, as constants in an array, so that obtaining any particular root of unity simply consists of determining the appropriate index into this array.

The primary method used for storing polynomials and power series is as a linked list of coefficients, using AlgolW's RECORD and REFERENCE facility, with no special distinction being made between dense and sparse polynomials and/or power series. To increase efficiency and to provide better control over the creation of such storage, AlgolW's dynamic RECORD allocation mechanism has been bypassed in favour of making direct system calls to obtain large blocks of storage which are then dispensed under greater control. Furthermore, because AlgolW has no storage reclamation facility, all storage is returned by each procedure to a central pool after it is no longer needed so that it can be re-used in later calculations.

When performing polynomial and power series arithmetic, the use of linked lists is inferior to a more directly accessible structure. Therefore, when performing any calculations, the coefficients of the polynomials or power series involved are transferred to an array structure until the operation is complete, after which the results are converted back into linked list form.

In order to obtain execution time values that are as independent as possible of the storage mechanism used, two different time values are obtained by each of the programs. The first value consists of the complete execution time for each Pade procedure. The second value is the time spent in storage manipulation for each Pade procedure. This latter value includes the time spent for polynomial and power

series storage allocation and reclamation as well as the time for converting from linked list format to array format and vice-versa. In this way, a non-storage specific time value can easily be determined. To prevent the time accumulation operation from interfering with the execution time values, each call to the standard AlgolW TIME procedure has been replaced with an SVC instruction (see Boettner and Alexander [3]) which returns the elapsed problem state time for the active task.

4.3 Empirical Results

In order to determine the practical efficiency of each of the four Pade algorithms, the implementation just described was used to obtain CPU time costs for each algorithm using a number of sample input power series. The programs were executed under the MTS operating system running on an Amdahl 470/V8 mainframe computer with a real memory space of 32 megabytes. The programs were executed with that execution as the only active task on the machine; no other user or system programs were operational at the time of execution. All of the background system tasks, such as the STAT job and LOADLEVL, were turned off in attempt to eliminate as much system interference as possible. To guarantee that there would be no interference in the results due to memory paging, the paging drum processor (PDP), which handles that particular system function, was also turned off, since a shortage of memory would not be a problem with

the available memory space.

The sample set of power series used to obtain the empirical CPU costs consisted of 18 power series with coefficients generated at random from the coefficient field, GF(40961). For 10 of these power series, the degree of conversion (the value of n when computing an (n,n) diagonal Pade approximant) was also chosen at random from between 0 and 250. The degree of conversion for the other 8 power series ranged from 250 to 2000 in increments of 250.

Each of the 18 power series was converted twice using each of the four Pade algorithms: once using classical polynomial and power series arithmetic and once using fast arithmetic, resulting in 8 total CPU time values and 8 associated storage manipulation time values for each sample power series. The actual CPU time values for each power series were then obtained by subtracting the storage manipulation time from the total CPU time value for each execution, producing a total of 8 actual CPU time values for each sample power series. These actual execution times are plotted, with the CPU time as a function of the degree of conversion, in the graph in Figure 1. In this graph, the solid lines represent the costs using classical polynomial and power series arithmetic while the broken lines represent the costs using fast arithmetic. For the sake of completeness, the actual CPU time values obtained, as well as the times spent by each program in storage manipulation, can be found in Appendix B at the end of this presentation.

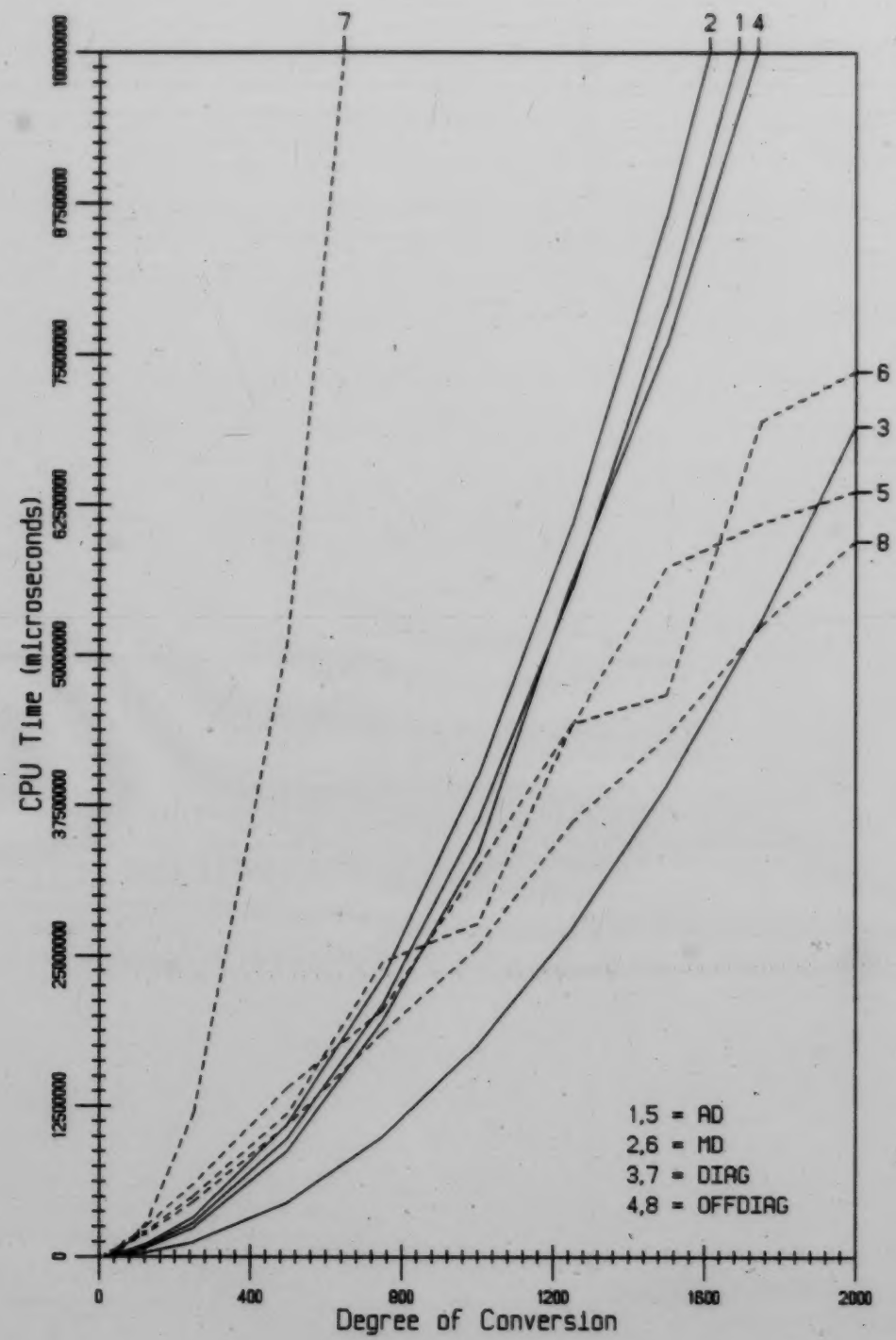

Figure 1: Empirical CPU Time Graph

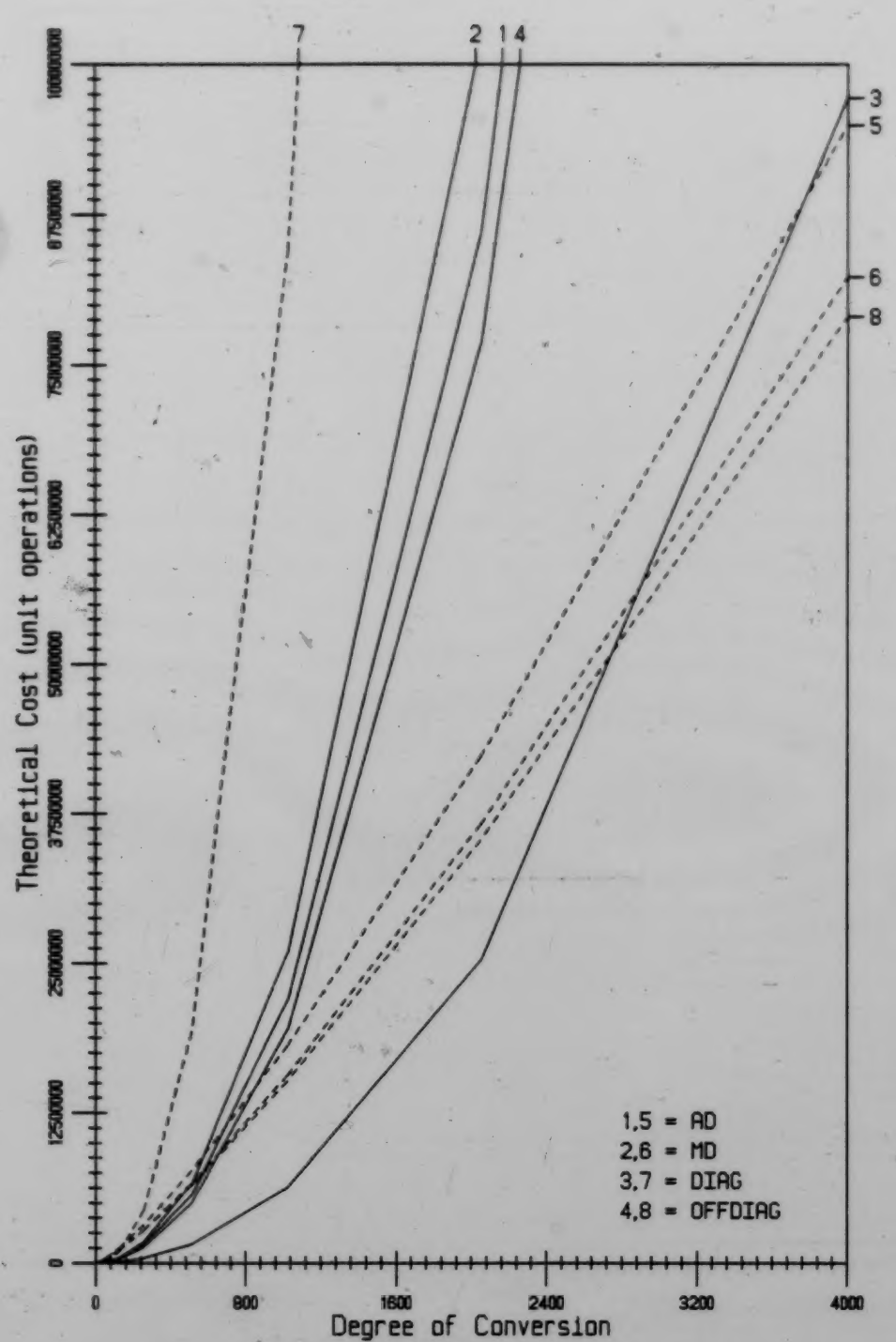

Figure 2: Theoretical Cost Graph

In Figure 2, a graph of the theoretical operation counts, plotted with the operation counts as a function of the degree of conversion, is also included to allow the making of comparisons. Solid lines are again used to indicate the use of classical arithmetic while broken lines again represent the use of fast arithmetic. A comparison of the two graphs indicate that, in terms of the general appearance of the graphs, there is little difference between the empirical behaviour and the theoretical behaviour of the four algorithms. A couple of noticable differences do exist between the two graphs, however, which can largely be attributed to two causes.

To begin with, the theoretical cost graph, as plotted, assumes that the value of n in each of the cost equations corresponds to the degree of conversion. While this is true when using classical arithmetic, the value of n when using fast arithmetic is actually closer to the nearest power of 2 greater than or equal to the degree of conversion. As a result, the lines in the theoretical cost graph corresponding to the use of fast arithmetic should more properly appear as staircases, rather than as smooth curves, indicating that the empirical CPU time graph is more indicative of the true behaviour of the Pade algorithms when using fast polynomial and power series arithmetic.

The second reason for differences between the two graphs is that there is an imbalance in the hidden costs, mostly in the form of procedure calls, that exist in the

implementation of the four algorithms. For example, in the implementation, each polynomial or power series operation requires a procedure call, the cost of which is not accounted for in any of the theoretical cost equations. Since the number of times that such costs are incurred is not the same for each of the algorithms, this has the effect of slightly altering the relations that exist between the four algorithms. In particular, it is for this reason that some of the crossover points in the empirical CPU time graph do not occur at the same place as they occur in the theoretical cost graph.

Up to this point, very little has been said about the storage manipulation times obtained. The reason for this is that the storage manipulation times represent only the costs of converting a polynomial or power series from a linked list structure to an array structure and back again. These costs are not an intrinsic property of the various algorithms and they would not exist if polynomials and power series are only stored using arrays. These time values are included in Appendix B simply to give an idea of how the use of linked lists affects the implementation.

4.4 Conclusions

As stated in the introduction, the ability to compute Pade approximants for a power series is useful to the efficient manipulation of rational functions in a symbolic computer algebra system. Motivated by this fact, a study was

undertaken to compare the best methods currently available for computing diagonal Pade approximants. Both the theoretical cost graph and the empirical cost graph that resulted from this study point to a very definite conclusion concerning the relative efficiencies of the four Pade algorithms considered.

It appears from both graphs that initially, for a substantial class of problems, the DIAG algorithm using classical polynomial and power series arithmetic is the best method for computing diagonal Pade approximants. Then, after the problem exceeds a certain size, the OFFDIAG algorithm using fast arithmetic takes over in this role. The other methods for computing diagonal Pade approximants are always inferior to at least one of these two approaches.

The actual point at which the use of OFFDIAG becomes superior to the use of DIAG varies depending on whether the empirical or the theoretical results are being used. In the case of the former, the crossover occurs at a degree of conversion of about 1700 while in the latter case, the degree of conversion for the crossover point is about 2700. For practical purposes, it is probably safest to use the empirical value as the crossover point in any implementation that combines the use of both methods (as proposed shortly) since both graphs indicate that the cost of the DIAG algorithm using classical arithmetic takes a sharp rise at this point, making it undesirable to keep using that approach for too long a period of time.

An immediate consequence of the high value of the crossover point given above is that it raises some question as to the usefulness of the OFFDIAG algorithm using fast arithmetic. It is not known whether there are any applications currently in use that require Pade approximants large enough for the OFFDIAG algorithm to become effective. In the absence of this information, however, it is probably not safe to eliminate the use of the OFFDIAG algorithm in a general system for handling rational functions using truncated power series until the intended applications are better known.

As a result of all this, it appears that the best approach to computing diagonal Pade approximants efficiently using any of the given four methods is to design an algorithm that takes advantage of the efficiencies of both the DIAG and the OFFDIAG techniques. Such an algorithm would initially employ the DIAG algorithm using classical arithmetic to compute consecutive Pade approximants along the main diagonal until a specific degree of conversion is reached. After this point, the OFFDIAG algorithm using fast arithmetic would be used to continue the process of computing approximants along the diagonal (although these would no longer be consecutive) until the desired one is reached. The recursive call made by the OFFDIAG procedure would not call OFFDIAG again immediately, as it currently does, but would instead call this combined procedure so that full advantage could be made of both algorithms in their

most efficient environment. Such a combined use of the DIAG and OFFDIAG algorithms would, quite naturally, be superior to the use of OFFDIAG only, and its crossover with the DIAG algorithm would be lower than the crossover of the DIAG and OFFDIAG algorithms. As a result, it becomes necessary to re-determine the point at which the combined algorithm becomes superior to the DIAG algorithm.

4.5 Thoughts for Future Research

The algorithms and results presented so far apply to power series and polynomials with coefficients that lie in a field. A substantial number of problems (perhaps even the majority) deal with polynomials and power series whose coefficients come from a more general algebraic system, most notably a unique factorization domain (ufd). It, therefore, becomes interesting to speculate as to whether these results can be extended to this larger class of polynomials and power series.

One way in which all four algorithms can be adapted to compute Pade approximants for a power series whose coefficients lie in a unique factorization domain is to make use of modular techniques. Using this approach, the initial power series is first mapped onto several Galois fields defined within the ufd using prime elements selected from the ufd. (Of course, Fourier primes should be selected in order to retain the ability to use Fourier-based fast arithmetic within the resulting fields.) The power series to

rational function conversion operation is then performed over each of these fields using the desired Pade algorithm. The results produced by each of these conversions are then combined using the Chinese Remainder Algorithm, as described by Lipson [17,19], to form a single solution over the given unique factorization domain.

The only costs introduced by the use of modular techniques is the cost of mapping the initial power series onto the selected Galois fields and the cost of the Chinese Remainder Algorithm. Since these costs are the same regardless of the method used to compute the Pade approximants, the relative efficiencies of the four Pade algorithms as applied to power series over a unique factorization domain using this approach remain the same as when the coefficient domain is a field.

A major disadvantage to the use of the Chinese Remainder Algorithm is that solutions over a potentially large number of different Galois fields may be required to produce the desired Pade approximant over the ufd. As an alternative to the use of this algorithm, it appears that decoupled p-adic construction via the Hensel Algorithm, a description of which is given by Yun [26], may be useful for mapping Pade approximants from a Galois field back to a ufd in much the same way as it is applied to the problem of polynomial division over a ufd. The advantage of using the p-adic technique is that only a single Galois field is required to produce the desired Pade approximant over the

given unique factorization domain. As with the use of the Chinese Remainder Algorithm, the relative efficiencies of the four Pade algorithms when applied over a unique factorization domain using p-adic construction would remain the same as when performed over a field.

A third, and perhaps the most obvious, way to extend the four given Pade algorithms to the class of power series over a ufd is to perform all power series and polynomial arithmetic directly over the desired unique factorization domain rather than over a field. Unfortunately, this approach suffers from a few difficulties. To begin with, since inverses of coefficients do not exist when the coefficient domain is a ufd, it is not possible to perform exact polynomial and power series division within this domain, making it necessary, instead, to use pseudo-division of polynomials and power series, as outlined, for example, by Knuth [16]. The problem with the use of pseudo-division is that it produces explosive coefficient growth in the results of calculations, a phenomenon well-known in the study of computer algebra. It is possible that the same technique used to ease this problem in the Subresultant PRS Algorithm for computing polynomial GCDs (see Knuth [16]) may also be used to reduce the rate of such coefficient growth in the Pade approximation algorithms, but this is not a certainty and requires further investigation.

A second difficulty with performing polynomial and power series arithmetic directly over a ufd is that

primitive roots of unity do not exist in a ufd so that fast polynomial and power series multiplication and, consequently, division based on the Fourier transform can not be performed directly over a unique factorization domain. Furthermore, an alternative method for performing fast polynomial and power series arithmetic over a ufd does not currently exist. It may be possible to employ modular techniques, using the Chinese Remainder Algorithm, to produce the desired multiplication result over a ufd from products computed using fast multiplication over several Galois fields properly chosen from within the ufd, but it is unclear as to whether the superior order of complexity of fast multiplication can be retained using this approach. Another possibility is that the polynomial or power series product can be computed over a single Galois field generated from within the ufd using a prime element that is at least twice as large as any coefficient in the result as produced over the ufd, bounds for which can easily be determined. If the proper residue class is used in this Galois field, then the product produced is the same as the result would be over the ufd. While the order of complexity for fast multiplication over a ufd remains the same using this approach as for over a field, it suffers from one difficulty in that, for the general case, where no practical maximum exists on the size of coefficients in the ufd, no single prime element is sufficient for producing any arbitrary product, making it necessary to continually find a new one,

along with a primitive root of unity for the field thus defined, for each multiplication problem, the cost of which is not trivial.

Another factor that needs to be considered when performing arithmetic directly over a ufd is that the cost of performing unit operations no longer remains a constant. Regardless of whether classical or fast arithmetic is being used, it inevitably becomes necessary, at a practical level, to perform indefinite precision arithmetic involving elements from the ufd since most unique factorization domains, such as the integers, contain a wealth of values that exceed the precision of most computers. Thus the cost of unit arithmetic in a ufd varies depending on the size of the elements being operated on.

In a slightly different context, a final question which should be raised is whether the results presented for the computation of Pade approximants for a power series over a field can still be improved. The answer to this is unclear. At present, no known work is being done on either the AD or the MD approaches to solving the Pade approximation problem. However, some study is being done to see if the power series division operations required by the DIAG and OFFDIAG algorithms can be removed in a manner similar to the way in which they are avoided by the MD algorithm. If this is possible, the costs of these two algorithms should improve dramatically since these operations represent the single largest contributor to the cost of these two methods,

particularly when using fast polynomial and power series arithmetic. Initial results in this area by Choi [7] indicate that the divisions can be eliminated for computing diagonal Pade approximants for a power series that is normal along the main diagonal of its Pade table. It will be interesting to see if this result can be extended to include the more general class of abnormal power series as well.

References

[1] Aho A., Hopcroft J., Ullman J., The Design and Analysis of Computer Algorithms, Addison-Wesley Publishing Co., Reading, Mass., 1974.

[2] Bentley J., Haken D., Saxe J., "A General Method for Solving Divide-and-Conquer Recurrences", ACM SIGACT News, 12, 3, Fall 1980, pp. 36-44.

[3] Boettner D., Alexander M., "The Michigan Terminal System", Proceedings of the IEEE, 63, 6, June 1975, pp. 912-918.

[4] Borodin A., Munro I., The Computational Complexity of Algebraic and Numeric Problems, American Elsevier Publishing Co., Inc., New York, N.Y., 1975.

[5] Brent R., Gustavson F., Yun D., "Fast Solution of Toeplitz Systems of Equations and Computation of Pade Approximants", Journal of Algorithms, 1, 3, September 1980, pp. 259-295.

[6] Cabay S., Kao T., "The Diagonal Pade Table and the Triangular Decomposition of Hankel Matrices", To appear.

[7] Choi D., PhD Thesis, University of Alberta, In preparation.

[8] Collins G., Computing Multiplicative Inverses in GF(p),

University of Wisconsin, Computer Sciences Technical Report No. 22, May, 1968.

[9] Fiduccia C., "Polynomial Evaluation via the Division Algorithm: The Fast Fourier Transform Revisited", Proceedings of the 4th Annual ACM Symposium on Theory of Computing, May 1 - 3, 1972, pp. 88-93.

[10] Geddes K., Algorithms for Analytic Approximation, University of Toronto, Computer Science Technical Report No. 56, August, 1973.

[11] Geddes K., "Symbolic Computation of Pade Approximants", ACM Transactions on Mathematical Software, 5, 2, June 1979, pp. 218-233.

[12] Gragg W., "The Pade Table and its Relation to Certain Algorithms of Numerical Analysis", SIAM Review, 14, 1, January 1972, pp. 1-62.

[13] Herstein I., Topics in Algebra, Second Edition, John Wiley and Sons, Inc., New York, N.Y., 1975.

[14] Horowitz E., Sahni S., Fundamentals of Computer Algorithms, Computer Science Press, Inc., Potomac, Maryland, 1978.

[15] Knuth D., Fundamental Algorithms, Second Edition, Addison-Wesley Publishing Co., Reading, Mass., 1973.

[16] Knuth D., Seminumerical Algorithms, Second Edition,

Addison-Wesley Publishing Co., Reading, Mass., 1981.

[17] Lipson J., "Chinese Remainder and Interpolation Algorithms", Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation, March 23 - 25, 1971, pp. 372-391.

[18] Lipson J., "Newton's Method: A Great Algebraic Algorithm", Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation, August 10 - 12, 1976, pp. 260-270.

[19] Lipson J., Elements of Algebra and Algebraic Computing, Addison-Wesley Publishing Co., Reading, Mass., 1981.

[20] McEliece R., Shearer J., "A Property of Euclid's Algorithm and an Application to Pade Approximation", SIAM Journal on Applied Mathematics, 34, 4, June 1978, pp. 611-615.

[21] Moenck R., "Fast Computation of GCDs", Proceedings of the 5th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1973, pp. 142-151.

[22] Moenck R., Studies in Fast Algebraic Algorithms, University of Toronto, Computer Science Technical Report No. 57, September, 1973.

[23] Moenck R., "Practical Fast Polynomial Multiplication", Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation, August 10 - 12, 1976,

pp. 136-148.

[24] MTS Manual, Volume 16: AlgolW in MTS, University of Michigan, Ann Arbor, Mich., 1980.

[25] Shockley J., Introduction to Number Theory, Holt, Rinehart and Winston, Inc., New York, N.Y., 1967.

[26] Yun D., "Algebraic Algorithms using p-adic Constructions", Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation, August 10 - 12, 1976, pp. 248-259.

Appendix A: Determining Theoretical Cost Estimates

A theoretical cost estimate for an algorithm is obtained by accumulating a count of the number of base operations, expressed in terms of the input size, required to execute the algorithm. In most cases, only the order of magnitude of such an estimate is required. However, in the present discussion, more accurate estimates are desired, making it necessary to have exact methods for counting unit operations. It is the purpose of this appendix to examine some of the techniques used for producing the cost estimates given in the current presentation.

The accumulation of the exact number of operations required to execute a linear sequence of statements in an algorithm is a straightforward, and simple, counting exercise. However, two algorithmic constructs, the loop and recursion, require somewhat more elaborate techniques to obtain the operation count.

To determine the total number of operations for executing a loop, it is necessary to be able to sum up a finite sequence of values. Normally, these sequences have a rigid fixed pattern to them. For the algorithms described in this presentation, four types of sequences occur.

Theorem A.0.1

Let a,n be arbitrary positive integers. Then

1) $\sum_{i=0}^{n-1} i = n(n-1)/2,$

$$2)\ \sum_{i=0}^{n-1} a^i = (a^n-1)/(a-1),$$

$$3)\ \sum_{i=0}^{n-1} ia^i = [(n-1)a^{n+1}-na^n+a]/(a-1)^2.$$

$$4)\ \sum_{i=0}^{n-1} a^{-i} = (a^n-1)/(a^n-a^{n-1}).$$

The correctness proofs for these four equations are simple inductive exercises. Two useful properties of such sums of sequences are given as follows:

Theorem A.0.2

Let f(i),g(i) be two functions with i as their input and let c be a constant. Then

$$1)\ \sum_{i=0}^{n-1} [f(i)+g(i)] = \sum_{i=0}^{n-1} f(i) + \sum_{i=0}^{n-1} g(i),$$

$$2)\ \sum_{i=0}^{n-1} cf(i) = c[\sum_{i=0}^{n-1} f(i)].$$

Again, the inductive proofs of the correctness of these equations are trivial. The results of these two theorems are all that is required for obtaining cost estimates for the iterative aspects of the algorithms in the present discussion.

In handling the recursive aspects of any algorithms, it is necessary to be able to convert the resulting recurrence relation into a more iterative equation of the type given by the previous two theorems. Two types of recurrence relations

need to be solved when determining cost estimates for the algorithms in the current discussion. The general solution to the first type of recurrence relation can be attributed to Bentley, Haken, and Saxe [2].

Theorem A.0.3

Assume $T(n) = kT(n/2)+f(n)$ where $n = 2^m$. Let $c = \log k$, $g(n) = f(n)/n^c$, and $h(n) = \sum_{i=1}^{\log n} g(2^i)$. Then

$$T(n) = n^c[T(1)+h(n)].$$

Proof:

The assertion holds by mathematical induction:

First, note that $k = k^{\log 2} = 2^{\log k} = 2^c$.

Therefore $T(n) = 2^cT(n/2)+f(n)$

$= 2^cT(n/2)+n^cg(n)$.

Note also that $h(n) = h(n/2)+g(n)$.

Assume $T(n) = n^c[T(1)+h(n)]$.

Then $T(2n) = kT(n)+f(2n)$

$= 2^cT(n)+(2n)^cg(2n)$

$= 2^c(n^c[T(1)+h(n)])+(2n)^cg(2n)$

$= (2n)^c[T(1)+h(n)+g(2n)]$

$= (2n)^c[T(1)+h(2n)]$.

Q.E.D.

Note that in the recurrence relation solved by this theorem, the value of n is assumed to be an exact power of 2, a situation that holds for all recurrence relations which occur in this presentation. The second recurrence relation

which needs to be solved is only required, in this presentation, by the cost equation for the OFFDIAG algorithm.

Theorem A.0.4

Assume $T(n) = \sum_{i=1}^{\log n} [T(2^{i-1})+f(2^i)]$ where $n = 2^m$. Then

$$T(n) = 2T(n/2)+f(n).$$

Proof:

By induction:

$$T(2n) = \sum_{i=1}^{\log 2n} [T(2^{i-1})+f(2^i)]$$

$$= T(n)+f(2n)+ \sum_{i=1}^{\log n} [T(2^{i-1})+f(2^i)]$$

$$= 2T(n)+f(2n).$$

Q.E.D.

In this second case, the recurrence relation is converted into another recurrence relation of the first type which can then be solved using the technique given by Theorem A.0.3.

Appendix B: Empirical CPU Time Values

For the sake of completeness, the actual CPU time values, obtained by subtracting the storage manipulation times from the total CPU times for each algorithm, as well as the times spent by each algorithm in storage manipulation, are presented in this appendix. In the four tables that follow, the first column, denoted by n, corresponds to the degree of conversion for each particular power series converted in the sample runs. Since the generation and implications of these time values are already discussed in Chapter 4 of this presentation, there is little more that needs to be said about them, so they are now given without any further comment.

n	CPU Times (microseconds)			
	AD	MD	DIAG	OFFDIAG
16	51459	70261	13646	32031
18	61823	81459	15313	40521
45	200078	251640	59531	137735
59	286563	361953	90365	207448
67	344818	431693	111823	280312
67	350078	428724	113073	280078
95	582473	707760	204193	449713
102	645546	783567	231692	506016
105	676562	816224	244114	526094
125	891302	1063802	332552	698229
250	2830234	3262760	1193594	2399636
500	9817812	10987719	4504976	8786250
750	20924296	23183398	9936544	19558483
1000	36160690	39795408	17501248	33483979
1250	55493593	60813327	27181328	56200435
1500	78974352	86350176	39014528	75693411
1750	106570704	116293440	52935664	101009578
2000	138284064	150663984	69040896	130504861

Table 1: CPU Times using Classical Arithmetic

n	CPU Times (microseconds)			
	AD	MD	DIAG	OFFDIAG
16	142969	191745	54349	125365
18	182213	232812	70052	171953
45	657526	666067	394609	525156
59	822578	1035885	620781	732708
67	1053177	1292239	821328	1024114
67	1074922	1294791	821249	1027656
95	1768515	1642083	1773958	1363150
102	1858515	1806953	2012421	1590130
105	1894270	2199687	2116484	1615156
125	2103724	2564843	2803880	1885052
250	5037083	6028047	11929375	4567474
500	11865924	14008039	51186512	10911640
750	24456065	20463037	134307728	18579000
1000	27622981	32232943	220060272	25668435
1250	44077103	44215637	393430288	36064308
1500	57273488	46702720	576299088	43146964
1750	60969040	69460016	759825664	52457729
2000	63629664	73588384	944027904	59502073

Table 2: CPU Times using Fast Arithmetic

n	Storage Manipulation Times (microseconds)			
	AD	MD	DIAG	OFFDIAG
16	85286	119583	26120	51250
18	104531	135312	33151	61953
45	307969	387292	157448	178906
59	427604	528646	260156	241380
67	490781	610078	331016	304974
67	493776	612969	330208	305495
95	754948	909687	641953	423490
102	806172	986823	735391	467083
105	833099	1021380	777448	482422
125	1040130	1244557	1084296	585104
250	2343203	2706458	4229270	1297942
500	5183411	5870937	16779984	2875703
750	8212968	9181770	38102592	4615677
1000	11441614	12691536	67986368	6341093
1250	14639687	16198385	106727696	8359869
1500	18089600	19865904	154027392	10107421
1750	21464000	23590048	210258320	11993958
2000	25068304	27381344	275289856	13872291

Table 3: Storage Manipulation Times using Classical Arithmetic

n	Storage Manipulation Times (microseconds)			
	AD	MD	DIAG	OFFDIAG
16	87682	122109	27292	52161
18	105469	137865	33568	64271
45	313177	399922	162214	183620
59	433099	544870	264948	250391
67	499401	631406	338281	317422
67	503854	629271	337865	313750
95	766953	931198	658958	439714
102	819818	1006901	752552	483984
105	846120	1047396	795729	494271
125	1054739	1273177	1102968	600000
250	2369687	2778385	4324843	1337213
500	5262604	6039817	17259568	2962734
750	8388255	9446979	39373328	4759192
1000	11639739	13099713	70123664	6546093
1250	14965729	16720859	110578928	8663020
1500	18501344	20471680	159600304	10458828
1750	21902704	24400624	217808896	12415807
2000	25505360	28271680	284588288	14343463

Table 4: Storage Manipulation Times using Fast Arithmetic